C++ Programming

Wikibooks.org

April 22, 2012

This PDF was generated by a program written by Dirk Hünniger, which is freely available under an open source license from HTTP://DE.WIKIBOOKS.ORG/WIKI/BENUTZER:DIRK_HUENNIGER/WB2PDF. The list of contributors is included in chapter Contributors on page 661. The licenses GPL, LGPL and GFDL are included in chapter Licenses on page 679, since this book and/or parts of it may or may not be licensed under one or more of these licenses, and thus require inclusion of these licenses. The licenses of the figures are given in the list of figures on page 675.

Contents

1 About the book **3**
- 1.1 Foreword . . . 3
- 1.2 Guide to readers . . . 3
- 1.3 Reader comments . . . 4

2 C++ a multi-paradigm language **7**
- 2.1 Introducing C++ . . . 7
- 2.2 What is a programming language? . . . 11
- 2.3 Programming paradigms . . . 16
- 2.4 Chapter summary . . . 40

3 Fundamentals for getting started **41**
- 3.1 The code . . . 41
- 3.2 The Compiler . . . 87
- 3.3 Variables . . . 121
- 3.4 Operators . . . 163
- 3.5 Type Conversion . . . 204
- 3.6 Control flow statements . . . 213
- 3.7 Functions . . . 229
- 3.8 Debugging . . . 367
- 3.9 Chapter Summary . . . 383

4 Object Oriented Programming **385**
- 4.1 Structures . . . 385
- 4.2 `union` . . . 390
- 4.3 Classes . . . 394
- 4.4 Copy Constructor . . . 436
- 4.5 Equality Operator . . . 436
- 4.6 Inequality Operator . . . 437
- 4.7 Operator overloading . . . 438
- 4.8 I/O . . . 451
- 4.9 Chapter Summary . . . 481

5 ADVANCED FEATURES **483**
5.1 TEMPLATES . . . 483
5.2 STANDARD TEMPLATE LIBRARY (STL) . . . 499
5.3 SMART POINTERS . . . 515
5.4 SEMANTICS . . . 516
5.5 EXCEPTION HANDLING . . . 517
5.6 RUN-TIME TYPE INFORMATION (RTTI) . . . 530
5.7 CHAPTER SUMMARY . . . 535

6 BEYOND THE STANDARD **537**
6.1 RESOURCE ACQUISITION IS INITIALIZATION (RAII) . . . 537
6.2 GARBAGE COLLECTION . . . 540
6.3 PROGRAMMING PATTERNS . . . 542
6.4 LIBRARIES . . . 584
6.5 BOOST LIBRARY . . . 588
6.6 CROSS-PLATFORM DEVELOPMENT . . . 598
6.7 SOFTWARE INTERNATIONALIZATION . . . 621
6.8 OPTIMIZATIONS . . . 628
6.9 FURTHER READING . . . 640
6.10 MODELING TOOLS . . . 640
6.11 CHAPTER SUMMARY . . . 641

7 APPENDIX A: INTERNAL REFERENCES **643**

8 APPENDIX B: EXTERNAL REFERENCES **645**
8.1 REFERENCE SITES . . . 645
8.2 COMPILERS AND IDES . . . 646
8.3 MISC. C++ TOOLS . . . 649
8.4 LIBRARIES[1] . . . 650
8.5 C++ CODING CONVENTIONS . . . 653
8.6 ONLINE C++ BOOKS, GUIDES AND GENERAL INFORMATION . . . 655
8.7 OTHER (DEAD TREE) BOOKS ON C++ . . . 660

9 CONTRIBUTORS **661**

LIST OF FIGURES **675**

10 LICENSES **679**
10.1 GNU GENERAL PUBLIC LICENSE . . . 679
10.2 GNU FREE DOCUMENTATION LICENSE . . . 680

1 Chapter 6.3.3 on page 584

10.3 GNU LESSER GENERAL PUBLIC LICENSE 681

1 About the book

1.1 Foreword

This book covers the C++ programming language, its interactions with software design and real life use of the language. It is presented as an introductory to advance course but can be used as reference book.

If you are familiar with programming in other languages you may just skim the **GETTING STARTED CHAPTER**[1]. You should not skip the **PROGRAMMING PARADIGMS SECTION**[2], because C++ does have some particulars that should be useful even if you already know another Object Oriented Programming language.

The **LANGUAGE COMPARISONS SECTION**[3] provides comparisons for some language(s) you may already know, which may be useful for veteran programmers.

If this is your first contact with programming then read the book from the beginning. Bear in mind that the *Programming Paradigms* section can be hard to digest if you lack some experience. Do not despair, the relevant points will be extended as other concepts are introduced. That section is provided so to give you a mental framework, not only to understand C++, but to let you easily adapt to (and from) other languages that may share concepts.

1.2 Guide to readers

This book is a WIKIBOOK[4] (EN.WIKIBOOKS.ORG)[5], an up-to-date copy of the work is hosted there.

1 Chapter 1.3 on page 5
2 Chapter 2.2.3 on page 16
3 Chapter 2.3.6 on page 22
4 HTTP://EN.WIKIPEDIA.ORG/WIKI/WIKIBOOK
5 HTTP://EN.WIKIBOOKS.ORG/WIKI/MAIN%20PAGE

It is organized into different parts, but as this is a work that is always evolving, things may be missing or just not where they should be, you are free to become a writer and contribute to fix things up...

1.3 Reader comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us (e.g. by using the "discussion" pages or by email). Be sure to include the section/title of the document with your comments and the date of your copy of the book. If you are really convinced of your point, information or correction then become a writer (at Wikibooks) and do it, it can always be rolled back if someone disagrees.

0^{6}

6 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

The following people are authors to this book:
PANIC[a], THENUB314[b]
You can verify who has contributed to this book by examining the history logs at Wikibooks (http://en.wikibooks.org/).
Acknowledgment is given for using some contents from other works like WIKIPEDIA[c], the wikibooks JAVA PROGRAMMING[d] and C PROGRAMMING[e] and the C++ REFERENCE[f] , as from the authors SCOTT WHEELER[g], STEPHEN FERG[h] and Ivor Horton .
The above authors release their work under the following license:
This work is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license. In short: you are free to share and to make derivatives of this work under the conditions that you appropriately attribute it, and that you only distribute it under the same, similar or a compatible license. Any of the above conditions can be waived if you get permission from the copyright holder. Unless otherwise noted, media and source code used in this book have their own copyright, may use different licenses than the one used here, and were not created by the above authors. The authors, contributors, and licenses used should be acknowledged separately.

a HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3APANIC2K4
b HTTP://EN.WIKIBOOKS.ORG/WIKI/USER%3ATHENUB314
c HTTP://EN.WIKIPEDIA.ORG/WIKI/
d HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING
e HTTP://EN.WIKIBOOKS.ORG/WIKI/C%20PROGRAMMING
f HTTP://WWW.CPPREFERENCE.COM
g HTTP://KTOWN.KDE.ORG/~{}WHEELER/BIO.HTML
h HTTP://WWW.FERG.ORG/INDEX.HTML

7

0^{8}

7 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
8 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

2 C++ a multi-paradigm language

2.1 Introducing C++

C++ (pronounced "see plus plus") is a *general-purpose*, *statically typed*, *free-form*, *multi-paradigm* PROGRAMMING LANGUAGE[1] supporting procedural programming, data abstraction, and generic programming. During the 1990s[2], **C++** became one of the most popular computer programming languages.

2.1.1 History and standardization



Figure 2: Photo of Bjarne Stroustrup, creator of the programming language C++.

1 Chapter 2.1.3 on page 11

2 HTTP://EN.WIKIPEDIA.ORG/WIKI/1990S

BJARNE STROUSTRUP[3], a Computer Scientist from BELL LABS[4], was the designer and original implementer of **C++** (originally named "C with Classes") during the 1980s as an enhancement to the C PROGRAMMING LANGUAGE[5]. Enhancements started with the addition OBJECT-ORIENTED[6] concepts like CLASSES[7], followed by, among many features, VIRTUAL FUNCTIONS[8], OPERATOR OVERLOADING[9], MULTIPLE INHERITANCE[10], TEMPLATES[11], and EXCEPTION HANDLING[12]. These and other features are covered in detail along this book.

The **C++ programming language** is a standard recognized by the ANSI[13] (The American National Standards Institute), BSI (The British Standards Institute), DIN (The German national standards organization), and several other national standards bodies, and was ratified in 1998 by the ISO (The International Standards Organization) as ISO/IEC 14882*[14]:1998*, consists of two parts: the Core Language and the Standard Library; the latter includes the STANDARD TEMPLATE LIBRARY[15] and the STANDARD C LIBRARY[16] (ANSI C 89).

Features introduced in **C++** include declarations as statements, function-like casts, `new`/`delete`, `bool`, reference types, `const`, `inline` functions, default arguments, function overloading, NAMESPACES[17], classes (including all class-related features such as inheritance, member functions, virtual functions, abstract classes, and constructors), operator overloading, templates, the `::` operator, exception handling, run-time type identification, and more type checking in several cases. Comments starting with two slashes ("`//`") were originally part of BCPL[18], and were reintroduced in **C++**. Several features of C++ were later adopted by C, including `const`, `inline`, declarations in `for` loops, and **C++**-style comments (using the `//` symbol).

---

3 HTTP://EN.WIKIPEDIA.ORG/WIKI/BJARNE%20STROUSTRUP
4 HTTP://EN.WIKIPEDIA.ORG/WIKI/BELL%20LABS
5 HTTP://EN.WIKIBOOKS.ORG/WIKI/SUBJECT%3AC%20PROGRAMMING%20LANGUAGE
6 Chapter 2.3.4 on page 19
7 Chapter 4.2.3 on page 393
8 Chapter 2.3.4 on page 21
9 Chapter 4.6 on page 438
10 Chapter 2.3.4 on page 20
11 Chapter 5 on page 483
12 Chapter 5.4 on page 517
13 HTTP://EN.WIKIPEDIA.ORG/WIKI/AMERICAN%20NATIONAL%20STANDARDS%20INSTITUTE
14 HTTP://EN.WIKIPEDIA.ORG/WIKI/ISO%2FIEC%2014882
15 Chapter 5.1.5 on page 499
16 Chapter 3.7.10 on page 264
17 Chapter 3.1.10 on page 79
18 HTTP://EN.WIKIPEDIA.ORG/WIKI/BCPL

The current version, which is the 2003 version, *ISO/IEC 14882:2003* redefines the standard language as a single item. The STL that pre-dated the standardization of **C++** and was originally implemented in Ada is now an integral part of the standard and a requirement for a compliant implementation of the same. Many other **C++** libraries exist which are not part of the Standard, such as BOOST[19]. Also, non-Standard libraries written in C can generally be used by **C++** programs.

Since 2004, the standards committee (which includes Bjarne Stroustrup) has been busy working out the details of a new revision of the standard, temporarily titled C++0x, due for publication in the end of 2011. Some implementations already support some of the proposed alterations.

C++ source code example

```
// 'Hello World!' program

#include <iostream>

int main()
{
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

Traditionally the first program people write in a new language is called "Hello World." because all it does is print the words **Hello World**. HELLO WORLD EXPLAINED[20] (in the EXAMPLES APPENDIX[21]) offers a detailed explanation of this code; the included source code is to give you an idea of a simple **C++** program.

2.1.2 Overview

Before you begin your journey to understand how to write programs using **C++**, it is important to understand a few key concepts that you may encounter. These concepts are not unique to **C++**, but are helpful to understanding computer programming in general. Readers who have experience in another programming language may wish to skim through this section entirely.

There are many different kinds of programs in use today. From the operating system you use that makes sure everything works as it should, to the video games and

19 Chapter 6.4.2 on page 588
20 Chapter 4.8.2 on page 457
21 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FEXAMPLES

music applications you use for fun, programs can fulfill many different purposes. What all **programs** (also called **software** or **applications**) have in common is that they all are made up of a sequence of instructions written in some form of programming language. These instructions tell a computer what to do, and generally how to do it. Programs can contain anything from instructions to solve math problems or send emails, to how to behave when a video game character is shot in a game. The computer will follow the instructions of a program one instruction at a time from start to finish.

2.1.3 Why learn C++ ?

Why not? This is the most clarifying approach to the decision to learn anything. Although learning is always good, selecting what you learn is more important as it is how you will prioritize tasks. Another side of this problem is that you will be investing some time in getting a new skill set. You must decide how this will benefit you. Check your objectives and compare similar projects or see what the programming market is in need of. In any case, the more programming languages you know, the better.

If you are approaching the learning process only to add another notch under your belt, that is, willing only to dedicate enough effort to understand its major quirks and learn something about its dark corners, then you would be best served in learning two other languages first. This will clarify what makes **C++** special in its approach to programming problems. You should select one imperative and one object-oriented language. C will probably be the best choice for the former, as it has a good market value and a direct relation to **C++**, although a good substitute would be ASM. Java is a good choice for the other language, for similar reasons.

If you are willing to dedicate a more than passing interest in **C++** then you can even learn it as your first language. Make sure to dedicate some time understanding the different paradigms and why C++ is a multi-paradigm, or hybrid, language.

Although learning C is not a requirement for understanding **C++**, you must know how to use an imperative language. **C++** will not make it easy for you to understand and distinguish some of these deeper concepts, since in it you are free to implement solutions with a greater range of freedom. Understanding which options to choose will become the cornerstone of mastering the language.

You should not learn **C++** if you are only interested in learning Object-oriented Programming, since the nomenclature used and some of the approaches taken to problems will make it more difficult to learn and master those concepts. If you are truly interested in Object-oriented programming, you should learn Smalltalk.

As with all languages, **C++** has a specific scope of application where it can truly shine. C++ is harder to learn than C and Java but more powerful than both. **C++** enables you to abstract from the little things you have to deal with in C or other lower level languages but will grant you more control and responsibility than Java. As it will not provide the default features you can obtain in similar higher level languages, you will have to search and examine several external implementations of those features and freely select those that best serve your purposes (or implement your own solution).

2.2 What is a programming language?

In the most basic terms, a "PROGRAMMING LANGUAGE[22]" is a means of communication between a human being (programmer) and a computer. A programmer uses this means of communication in order to give the computer instructions. These instructions are called "programs".

Like the many languages we use to communicate with each other, there are many languages that a programmer can use to communicate with a computer. Each language has its own set of words and rules, called semantics. If you're going to write a program, you have to follow the semantics of the language you're writing in, or you won't be understood.

Programming languages can basically be divided in to two categories: LOW-LEVEL[23] and HIGH-LEVEL[24], next we will introduce you to these concepts and their relevance to C++.

22 HTTP://EN.WIKIPEDIA.ORG/WIKI/PROGRAMMING%20LANGUAGE

23 HTTP://EN.WIKIPEDIA.ORG/WIKI/LOW-LEVEL%20PROGRAMMING%20LANGUAGE

24 HTTP://EN.WIKIPEDIA.ORG/WIKI/HIGH-LEVEL%20PROGRAMMING%20LANGUAGE

2.2.1 Low-level

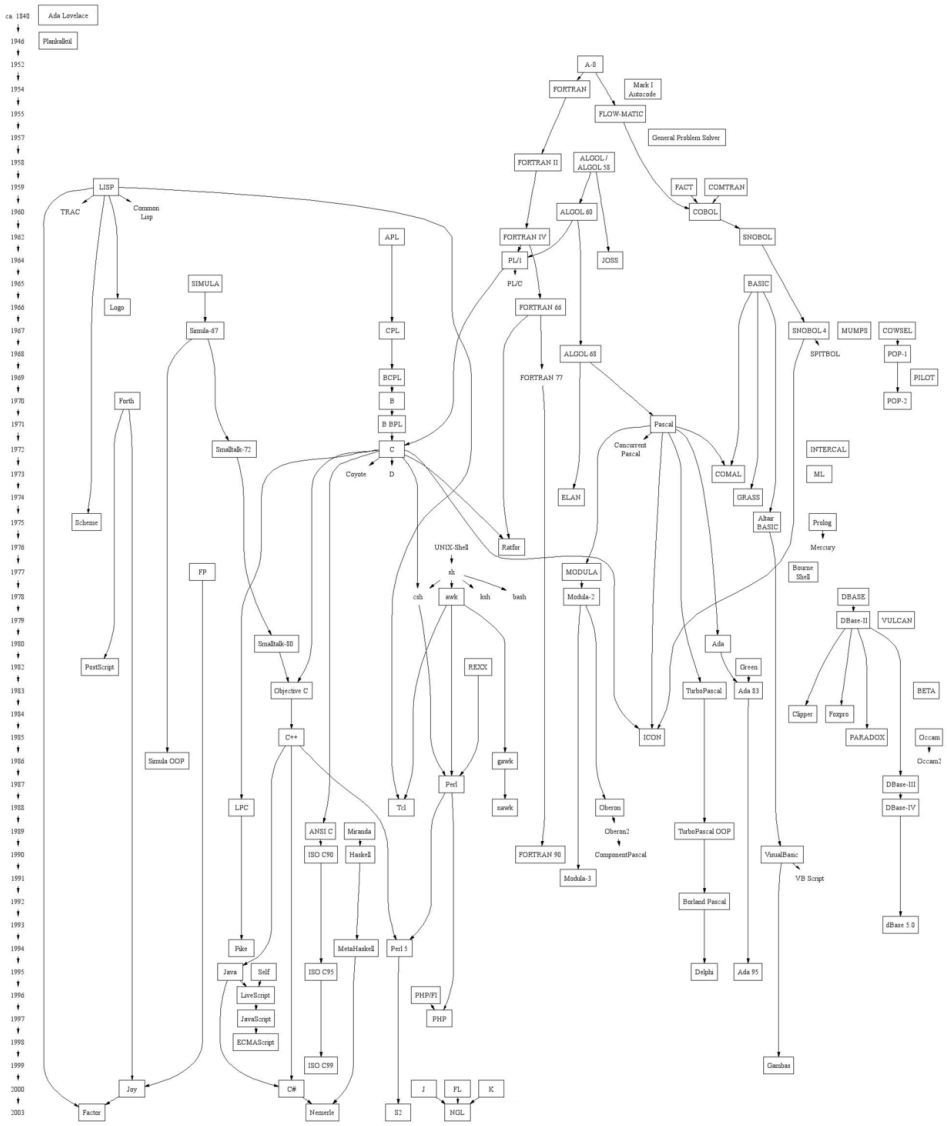


Figure 3: Image shows most programming languages and their relations from mid 18 hundreds up to 2003 (CLICK HERE FOR FULL SIZE[a]).

a HTTP://EN.WIKIBOOKS.ORG/WIKI/MEDIA%3ATAXONOMYOFPROGRAMMINGLANGUAGES.PNG

The lower level in computer "languages" are:

Machine code (also called binary) is the lowest form of a low-level language. Machine code consists of a string of 0s and 1s, which combine to form meaningful instructions that computers can take action on. If you look at a page of binary it becomes apparent why binary is never a practical choice for writing programs; what kind of person would actually be able to remember what a bunch of strings of 1 and 0 mean?

Assembly language (also called ASM), is just above machine code on the scale from low level to high level. It is a human-readable translation of the machine language instructions the computer executes. For example, instead of referring to processor instructions by their binary representation (0s and 1s), the programmer refers to those instructions using a more memorable (mnemonic) form. These mnemonics are usually short collections of letters that symbolize the action of the respective instruction, such as "ADD" for addition, and "MOV" for moving values from one place to another.

Note:
Assembly language is *processor specific*. This means that a program written in assembly language will not work on computers with different processor architectures.

Using ASM to optimize certain tasks is common for C++ programmers, but will require special considerations, because ASM is not as portable.

You do not have to understand assembly language to program in C++, but it does help to have an idea of what's going on "behind-the-scenes". Learning about assembly language will also allow you to have more control as a programmer and help you in debugging and understanding code.

The advantages of writing in a high-level language format far outweigh any drawbacks, due to the size and complexity of most programming tasks, those advantages include:

- Advanced program structure: loops, functions, and objects all have limited usability in low-level languages, as their existence is already considered a "high" level feature; that is, each structure element must be further translated into low-level language.
- Portability: high-level programs can run on different kinds of computers with few or no modifications. Low-level programs often use specialized functions available on only certain processors, and have to be rewritten to run on another computer.

- Ease of use: many tasks that would take many lines of code in assembly can be simplified to several function calls from libraries in high-level programming languages. For example, Java, a high-level programming language, is capable of painting a functional window with about five lines of code, while the equivalent assembly language would take at least four times that amount.

2.2.2 High-level

High-level languages do more with less code, although there is sometimes a loss in performance and less freedom for the programmer. They also attempt to use English language words in a form which can be read and generally interpreted by the average person with little experience in them. A program written in one of these languages is sometimes referred to as "human-readable code". In general, more abstraction makes it easier for a language be learned.

No programming language is written in what one might call "plain English" though, (although BASIC comes close). Because of this, the text of a program is sometimes referred to as "code", or more specifically as "source code." This is discussed in more detail in the THE CODE SECTION[25] of the book.

Higher-level languages partially solve the problem of abstraction to the hardware (CPU, co-processors, number of registers etc...) by providing portability of code.

Keep in mind that this classification scheme is evolving. C++ is still considered a high-level language, but with the appearance of newer languages (Java, C#, Ruby etc...), C++ is beginning to be grouped with lower level languages like C.

2.2.3 Translating programming languages

Since a computer is only capable of understanding machine code, human-readable code must be either interpreted or translated into machine code.

An **INTERPRETER**[26] is a program (often written in a lower level language) that interprets the instructions of a program one instruction at a time into commands that are to be carried out by the interpreter as it happens. Typically each instruction consists of one line of text or provides some other clear means of telling each instruction apart and the program must be reinterpreted again each time the program is run.

25 Chapter 3 on page 41

26 HTTP://EN.WIKIPEDIA.ORG/WIKI/INTERPRETER%20%28COMPUTING%29

A **COMPILER**[27] is a program used to translate the source code, one instruction at a time, into machine code. The translation into machine code may involve splitting one instruction understood by the compiler into multiple machine instructions. The instructions are only translated once and after that the machine can understand and follow the instructions directly whenever it is instructed to do so. A complete examination of the C++ compiler is given in the COMPILER SECTION[28] of the book.

The words and statements used to instruct the computer may differ, but no matter what words and statements are used, just about every programming language will include statements that will accomplish the following:

Input

Input is the act of getting information from a device such as a keyboard or mouse, or sometimes another program.

Output

Output is the opposite of input; it gives information to the computer monitor or another device or program.

Math/Algorithm

All computer processors (the brain of the computer), have the ability to perform basic mathematical computation, and every programming language has some way of telling it to do so.

Testing

Testing involves telling the computer to check for a certain condition and to do something when that condition is true or false. Conditionals are one of the most important concepts in programming, and all languages have some method of testing conditions.

Repetition

Perform some action repeatedly, usually with some variation.

An further examination is provided on the STATEMENTS SECTION[29] of the book.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of functions that look more or less

27 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPILER
28 Chapter 3.1.10 on page 87
29 Chapter 3.1.6 on page 56

like these. Thus, one way to describe programming is the process of breaking a large, complex task up into smaller and smaller subtasks until eventually the subtasks are simple enough to be performed with one of these simple functions.

C++ is mostly *compiled* rather than *interpreted* (there are some C++ interpreters), and then "executed" later. As complicated as this may seem, later you will see how easy it really is.

So as we have seen in the INTRODUCING C++ SECTION[30], C++ evolved from C by adding some levels of abstraction (so we can correctly state that C++ is of a higher level than C). We will learn the particulars of those differences in the PROGRAMMING PARADIGMS SECTION[31] of the book and for some of you that already know some other languages should look into PROGRAMMING LANGUAGES COMPARISONS SECTION[32].

2.3 Programming paradigms

A **PROGRAMMING PARADIGM**[33] is a model of programming based on distinct concepts that shapes the way programmers design, organize and write programs. A MULTI-PARADIGM PROGRAMMING LANGUAGE[34] allows programmers to choose a specific single approach or mix parts of different programming paradigms. C++ as a multi-paradigm programming language supports single or mixed approaches using Procedural or Object-oriented programming and mixing in utilizations of Generic and even Functional programming concepts.

2.3.1 Procedural programming

PROCEDURAL PROGRAMMING[35] can be defined as a subtype of IMPERATIVE PROGRAMMING[36] as a programming paradigm based upon the concept of procedure calls, in which STATEMENTS[37] are structured into procedures (also known

30 Chapter 2 on page 7
31 Chapter 2.2.3 on page 16
32 Chapter 2.3.6 on page 22
33 HTTP://EN.WIKIPEDIA.ORG/WIKI/PROGRAMMING%20PARADIGM
34 HTTP://EN.WIKIPEDIA.ORG/WIKI/MULTIPARADIGM%20PROGRAMMING%20LANGUAGE
35 HTTP://EN.WIKIPEDIA.ORG/WIKI/PROCEDURAL%20PROGRAMMING
36 HTTP://EN.WIKIPEDIA.ORG/WIKI/IMPERATIVE%20PROGRAMMING
37 Chapter 3.1.6 on page 56

as subroutines or FUNCTIONS[38]). Procedure calls are modular and are bound by scope. A procedural program is composed of one or more MODULES[39]. Each module is composed of one or more SUBPROGRAMS[40]. Modules may consist of procedures, functions, subroutines or methods, depending on the programming language. Procedural programs may possibly have multiple levels or scopes, with subprograms defined inside other subprograms. Each scope can contain names which cannot be seen in outer scopes.

Procedural programming offers many benefits over simple sequential programming since procedural code:

- is easier to read and more maintainable
- is more flexible
- facilitates the practice of good program design
- allows modules to be reused in the form of CODE LIBRARIES[41].

Note:
Nowadays it is very rare to see C++ strictly using the Procedural Programming paradigm, mostly it is used only on small demonstration or test programs.

2.3.2 Statically typed

Typing refers to how a computer language handles its variables, how they are differentiated by TYPE[42]. Variables are values that the program uses during execution. These values can change; they are variable, hence their name. *Static typing* usually results in compiled code that executes more quickly. When the compiler knows the exact types that are in use, it can produce machine code that does the right thing easier. In C++, variables need to be defined before they are used so that compilers know what type they are, and hence is statically typed. Languages that are not statically typed are called *dynamically typed.*

Static typing usually finds type errors more reliably at compile time, increasing the reliability of compiled programs. Simply put, it means that "A round peg won't fit in a square hole", so the compiler will report it when a type leads to ambiguity or incompatible usage. However, programmers disagree over how common type

38 Chapter 3.6.3 on page 229
39 HTTP://EN.WIKIPEDIA.ORG/WIKI/MODULE%20%28PROGRAMMING%29
40 HTTP://EN.WIKIPEDIA.ORG/WIKI/SUBPROGRAM%20%28PROGRAMMING%29
41 Chapter 6.3.3 on page 584
42 Chapter 3.3.3 on page 138

errors are and what proportion of bugs that are written would be caught by static typing. Static typing advocates believe programs are more reliable when they have been type checked, while dynamic typing advocates point to dynamic code that has proved reliable and to small bug databases. The value of static typing, then, presumably increases as the strength of the type system is increased.

A statically typed system constrains the use of powerful language constructs more than it constrains less powerful ones. This makes powerful constructs harder to use, and thus places the burden of choosing the "right tool for the problem" on the shoulders of the programmer, who might otherwise be inclined to use the most powerful tool available. Choosing overly powerful tools may cause additional performance, reliability or correctness problems, because there are THEORETICAL LIMITS[43] on the properties that can be expected from powerful language constructs. For example, indiscriminate use of RECURSION[44] or GLOBAL VARIABLE[45]s may cause well-documented adverse effects.

Static typing allows construction of libraries which are less likely to be accidentally misused by their users. This can be used as an additional mechanism for communicating the intentions of the library developer.

2.3.3 Type checking

Type checking is the process of verifying and enforcing the constraints of types, which can occur at either compile-time or run-time. Compile time checking, also called *static type* checking, is carried out by the compiler when a program is compiled. Run time checking, also called *dynamic type checking*, is carried out by the program as it is running. A programming language is said to be *strongly typed* if the type system ensures that conversions between types must be either valid or result in an error. A *weakly typed* language on the other hand makes no such guarantees and generally allows automatic conversions between types which may have no useful purpose. C++ falls somewhere in the middle, allowing a mix of automatic type conversion and programmer defined conversions, allowing for almost complete flexibility in interpreting one type as being of another type. Converting variables or expression of one type into another type is called TYPE CASTING[46].

43 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPUTATIONAL%20COMPLEXITY%20THEORY

44 HTTP://EN.WIKIPEDIA.ORG/WIKI/RECURSION

45 HTTP://EN.WIKIPEDIA.ORG/WIKI/GLOBAL%20VARIABLE

46 Chapter 3.4.14 on page 204

2.3.4 Object-oriented programming

OBJECT-ORIENTED PROGRAMMING[47] can be seen as an extension of procedural programming in which programs are made up of collection of individual units called **objects** that have a distinct purpose and function with limited or no dependencies on IMPLEMENTATION[48]. For example, a car is like an object; it gets you from point A to point B with no need to know what type of engine the car uses or how the engine works. Object-oriented languages usually provide a means of DOCUMENTING[49] what an object can and cannot do, like instructions for driving a car.

Objects and Classes

An **object** is composed of **members** and **methods**. The members (also called *data members*, *characteristics*, *attributes*, or *properties*) describe the object. The methods generally describe the actions associated with a particular object. Think of an object as a noun, its members as adjectives describing that noun, and its methods as the verbs that can be performed by or on that noun.

For example, a sports car is an object. Some of its members might be its height, weight, acceleration, and speed. An object's members just hold data about that object. Some of the methods of the sports car could be "drive", "park", "race", etc. The methods really do not mean much unless associated with the sports car, and the same goes for the members.

The blueprint that lets us build our sports car object is called a **class**. A class does not tell us how fast our sports car goes, or what color it is, but it does tell us that our sports car will have a member representing speed and color, and that they will be say, a number and a word, respectively. The class also lays out the methods for us, telling the car how to park and drive, but these methods can not take any action with just the blueprint - they need an object to have an effect.

Encapsulation

> *«No component in a complex system should depend on the internal details of any other component.»*

47 HTTP://EN.WIKIPEDIA.ORG/WIKI/OBJECT-ORIENTED%20PROGRAMMING
48 HTTP://EN.WIKIPEDIA.ORG/WIKI/IMPLEMENTATION
49 HTTP://EN.WIKIPEDIA.ORG/WIKI/DOCUMENTATION

--Dan Ingalls (Smalltalk Architect)

Encapsulation, the principle of INFORMATION HIDING[50] (from the user), is the process of hiding the data structures of the class and allowing changes in the data through a public interface where the incoming values are checked for validity, and so not only it permits the hiding of data in an object but also of behavior. This prevents clients of an interface from depending on those parts of the implementation that are likely to change in future, thereby allowing those changes to be made more easily, that is, without changes to clients. In modern programming languages, the principle of information hiding manifests itself in a number of ways, including encapsulation and polymorphism.

Inheritance

INHERITANCE[51] describes a relationship between two (or more) types, or classes, of objects in which one is said to be a "subtype" or "child" of the other, as result the "child" object is said to *inherit* features of the parent, allowing for shared functionality, this lets programmers re-use or reduce code and simplifies the development and maintenance of software.

Inheritance is also commonly held to include *subtyping*, whereby one type of object is defined to be a more specialized version of another type (see LISKOV SUBSTITUTION PRINCIPLE[52]), though non sub-typing inheritance is also possible.

Inheritance is typically expressed by describing *classes* of objects arranged in an *inheritance hierarchy* (also referred to as *inheritance chain*), a the tree like structure created by their inheritance relationships.

For example, one might create a variable class "Mammal" with features such as eating, reproducing, etc.; then define a subtype "Cat" that inherits those features without having to explicitly program them, while adding new features like "chasing mice". This allows commonalities among different kinds of objects to be expressed once and reused multiple times.

In C++ we can then have classes that are related to other classes (a class can be defined by means of an older, pre-existing, `class`). This leads to a situation in which a new class has all the functionality of the older class, and additionally introduces

50 HTTP://EN.WIKIPEDIA.ORG/WIKI/INFORMATION%20HIDING

51 HTTP://EN.WIKIPEDIA.ORG/WIKI/INHERITANCE%20%28OBJECT-ORIENTED%20PROGRAMMING%29

52 HTTP://EN.WIKIPEDIA.ORG/WIKI/LISKOV%20SUBSTITUTION%20PRINCIPLE

its own specific functionality. Instead of composition, where a given class contains another class, we mean here derivation, where a given class is another class.

This OOP property will be explained further when we talk about Classes (and Structures) inheritance in the CLASSES INHERITANCE SECTION[53] of the book.

If one wants to use more than one totally orthogonal hierarchy simultaneously, such as allowing "Cat" to inherit from "Cartoon character" and "Pet" as well as "Mammal" we are using MULTIPLE INHERITANCE[54].

Multiple inheritance

Multiple inheritance is the process by which one class can inherit the properties of two or more classes (variously known as its base classes, or parent classes, or ancestor classes, or super-classes).

In some similar language, multiple inheritance is restricted in various ways to keep the language simple, such as by allowing inheritance from only one real class and a number of "interfaces", or by completely disallowing multiple inheritance. C++ places the full power of multiple inheritance in the hands of programmers, but it is needed only rarely, and (as with most techniques) can complicate code if used inappropriately. Because of C++'s approach to multiple inheritance, C++ has no need of separate language facilities for "interfaces"; C++'s classes can do everything that interfaces do in some related languages.

This is shown more in more detail in the C++ CLASSES INHERITANCE SECTION[55] of the book.

Polymorphism

Polymorphism allows a single name to be reused for several related but different purposes. The purpose of polymorphism is to allow one name to be used for a general class. Depending on the type of data, a specific instance of the general case is executed.

The concept of polymorphism is wider. Polymorphism exists every time we use two functions that have the same name, but differ in the implementation. They may also differ in their interface, e.g., by taking different arguments. In that case the choice of which function to make is via overload resolution, and is performed at compile time, so we refer to *static polymorphism*.

53 Chapter 4.3.2 on page 398
54 Chapter 2.3.4 on page 21
55 Chapter 4.3.2 on page 398

Dynamic polymorphism will be covered deeply in the CLASSES SECTION[56] where we will address its use on redefining the method in the derived class.

2.3.5 Generic programming

GENERIC PROGRAMMING[57] or **POLYMORPHISM**[58] is a programming style that emphasizes techniques that allow one value to take on different types as long as certain contracts such as SUBTYPES[59] and SIGNATURE[60] are kept. In simpler terms generic programming is based in finding the most abstract representations of efficient algorithms. TEMPLATES[61] popularized the notion of generics. Templates allow code to be written without consideration of the TYPE[62] with which it will eventually be used. Templates are defined in the STANDARD TEMPLATE LIBRARY (STL)[63], where generic programming was introduced into C++.

2.3.6 Free-form

Free-form refers to how the programmer crafts the code. Basically, there are no rules on how you choose to write your program, save for the semantic rules of C++. Any C++ program should compile as long as it is legal C++.

The free-form nature of C++ is used (or abused, depending on your point of view) by some programmers in crafting obfuscated C++ (C++ that is purposefully written to be difficult to understand). The use of obfuscation is regarded by some as a security mechanism, ensuring that the source code is more difficult to analyze by the average user or to prevent the functionality from being duplicated.

2.3.7 Language comparisons

There is not a perfect language. It all depends on the resources (tools, people even available time) and the objective. For a broader look on other languages and their

56 Chapter 4.3.5 on page 418
57 HTTP://EN.WIKIPEDIA.ORG/WIKI/GENERIC%20PROGRAMMING
58 HTTP://EN.WIKIPEDIA.ORG/WIKI/POLYMORPHISM%20%28COMPUTER%20SCIENCE%29
59 HTTP://EN.WIKIPEDIA.ORG/WIKI/SUBTYPE
60 HTTP://EN.WIKIPEDIA.ORG/WIKI/SIGNATURE%20%28COMPUTER%20SCIENCE%29
61 HTTP://EN.WIKIPEDIA.ORG/WIKI/TEMPLATE%20%28PROGRAMMING%29
62 HTTP://EN.WIKIPEDIA.ORG/WIKI/DATATYPE
63 Chapter 5.1.5 on page 499

evolution, a subject that falls outside of the scope of this book, there are many other works available, including the COMPUTER PROGRAMMING[64] wikibook.

This section is provided as a quick jump-start for people that already had some experience in them, a way to edify notions about C++ language special characteristics and what makes it distinct.

Ideal language

The ideal language depends on the specific problem. All programming languages are designed to be *general mechanisms for expressing problem solving algorithms*. In other words, it is a language - rather than simply an expression - because it is capable of expressing solutions more than one specific problem.

The level of generality in a programming language varies. There are DOMAIN-SPECIFIC LANGUAGES[65] (DSLs) such as regular expression syntax which is designed specifically for pattern matching and string manipulation problems. There are also general-purpose programming languages such as C++.

Ultimately, there is no perfect language. There are some languages that are more suited to specific classes of problems than others. Each language makes tradeoffs, favoring efficiency in one area for inefficiencies in other areas. Furthermore, efficiency may not only mean *runtime performance* but also includes factors such as development time, code maintainability, and other considerations that affect software development. The best language is dependent on the specific objectives of the programmers.

Furthermore, another very practical consideration when selecting a language is the number and quality of tools available to the programmer for that language. No matter how good a language is in theory, if there is no set of reliable tools on the desired platform, that language is not the best choice.

The optimal language (in terms of run-time performance) is machine code but MACHINE CODE[66] (binary) is the least efficient programming language in terms of coder time. The complexity of writing large systems is enormous with high-level languages, and beyond human capabilities with machine code. In the next sections

64 HTTP://EN.WIKIBOOKS.ORG/WIKI/COMPUTER%20PROGRAMMING
65 HTTP://EN.WIKIPEDIA.ORG/WIKI/DOMAIN-SPECIFIC_LANGUAGE
66 HTTP://EN.WIKIPEDIA.ORG/WIKI/MACHINE%20CODE

C++ will be compared with other closely related languages like C[67], JAVA[68], C#[69], C++/CLI[70] and D[71].

«When someone says "I want a programming language in which I need only say what I wish done," give him a lollipop.»
--published in SIGPLAN Notices Vol. 17, No. 9, September 1982

The quote above is shown to indicate that no programming language at present can translate directly concepts or ideas into useful code, there are solutions that will help. We will cover the use of COMPUTER-AIDED SOFTWARE ENGINEERING (CASE)[72] tools that will address part of this problem but its use does require planning and some degree of complexity.

The intention of these sections is not to promote one language above another; each has its applicability. Some are better in specific tasks, some are simpler to learn, others only provide a better level of control to the programmer. This all may depend also on the level of control the programmer has of a given language.

Garbage collection

In C++ garbage collection is optional rather than required. In the GARBAGE COLLECTION SECTION[73] of this book we will cover this issue deeply.

Why no `finally` keyword?

As we will see in the RESOURCE ACQUISITION IS INITIALIZATION (RAII) SECTION[74] of the book, RAII can be used to provide a better solution for most issues. When `finally` is used to clean up, it has to be written by the clients of a class each time that class is used (for example, clients of a *fileClass class* have to do I/O in a `try`/`catch`/`finally` block so that they can guarantee that the *fileClass* is closed). With RAII, the destructor of the *fileClass* can make that guarantee. Now

67 Chapter 2.3.7 on page 25
68 Chapter 2.3.7 on page 27
69 Chapter 2.3.7 on page 37
70 Chapter 2.3.7 on page 39
71 Chapter 2.3.7 on page 39
72 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPUTER-AIDED%20SOFTWARE%20ENGINEERING%20%28CASE%29
73 Chapter 6.1 on page 540
74 Chapter 6 on page 537

the cleanup code has to be coded only once — in the destructor of *fileClass*; the users of the class don't need to do anything.

Mixing languages

By default, the C++ compiler normally "mangles" the names of functions in order to facilitate function overloading and generic functions. In some cases, you need to gain access to a function that wasn't created in a C++ compiler. For this to occur, you need to use the `extern` keyword to declare that function as external:

```
extern "C" void LibraryFunction();
```

C 89/99

C[75] was essentially the core language of C++ when Bjarne Stroustrup decided to create a "better C". Many of the syntax conventions and rules still hold true, so we can even state that C was a subset of C++. Most recent C++ compilers can also compile C code, taking into consideration the small incompatibilities, since C99[76] and C++ 2003 are not compatible any more. You can also check more information about the C language on the C PROGRAMMING WIKIBOOK[77].

Note:
In practice, much C99 code will still compile with a C++ compiler, but the language is no longer a proper subset. Compatibility is not guaranteed.

C++ as defined by the ANSI standard in 1998 (called C++98 at times) is very nearly, but not quite, a superset of the C language as it was defined by its first ANSI standard in 1989 (known as C89). There are a number of ways in which C++ is not a strict superset, in the sense that not all valid C89 programs are valid C++ programs, but the process of converting C code to valid C++ code is fairly trivial (avoiding reserved words, getting around the stricter C++ type checking with casts, declaring every called function, and so on).

In 1999, C was revised and many new features were added to it. As of 2004, most of these new "C99" features are not in C++. Some (including Stroustrup himself) have argued that the changes brought about in C99 have a philosophy

75 HTTP://EN.WIKIBOOKS.ORG/WIKI/SUBJECT%3AC%20PROGRAMMING%20LANGUAGE
76 HTTP://EN.WIKIPEDIA.ORG/WIKI/C99
77 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%20PROGRAMMING

distinct from what C++98 adds to C89, and hence these C99 changes are directed towards increasing incompatibility between C and C++.

The merging of the languages seems a dead issue, as coordinated actions by the C and C++ standards committees leading to a practical result did not happen and it can be said that the languages started to diverge.

Some of the differences are:

- C++ supports function overloading, this is absent in C, especially in C89 (it can be argued, depending on how loosely function overloading is defined, that it is possible to some degree to emulate these capabilities using the C99[78] standard).
- C++ supports INHERITANCE[79] and POLYMORPHISM[80].
- C++ adds keyword **class**, but keeps **struct** from C, with compatible semantics.
- C++ supports access control for class members.
- C++ supports generic programming through the use of TEMPLATES[81].
- C++ extends the C89 standard library with its own standard library.
- C++ and C99 offer different complex number facilities.
- C++ has **bool** and **wchar_t** as primitive types, while in C they are typedefs.
- C++ comparison operators returns **bool**, while C returns **int**.
- C++ supports overloading of operators.
- C++ character constants have type **char**, while C character constants have type **int**.
- C++ has specific CAST OPERATORS[82] (`static_cast`, `dynamic_cast`, `const_cast` and `reinterpret_cast`).
- C++ adds **mutable** keyword to address the imperfect match between physical and logical constness.
- C++ extends the type system with *references*.
- C++ supports **member functions**, **constructors** and **destructors** for user-defined types to establish invariants and to manage resources.
- C++ supports RUNTIME TYPE IDENTIFICATION[83] (RTTI), via **typeid** and `dynamic_cast`.
- C++ includes EXCEPTION HANDLING[84].
- C++ has `std::vector` as part of its standard library instead of variable-length arrays as in C.

78 HTTP://EN.WIKIPEDIA.ORG/WIKI/C99
79 Chapter 2.3.4 on page 20
80 Chapter 2.3.4 on page 21
81 Chapter 5 on page 483
82 Chapter 3.4.14 on page 204
83 Chapter 5.5.5 on page 530
84 Chapter 5.4 on page 517

- C++ treats `sizeof` operator as compile time operation, while C allows it be a runtime operation.
- C++ has **new** and **delete** operators, while C uses **malloc** and **free** library functions.
- C++ supports object-oriented programming without extensions.
- C++ does not require use of macros, unlike C, that uses them for careful information-hiding and abstraction (especially important for C code portability).
- C++ supports per-line comments denoted by **//**. (C99 started official support for this comment system, and most compilers support this as an extension.)
- C++ `register` keyword is semantically different to C's implementation.

Choosing C or C++

It is not uncommon to find someone defending C over C++ (or vice versa) or complaining about some features of these languages. There is no scientific evidence to put a language above another in general terms; the only reason that does have some traction is the possibility of deep changes or unknown bugs in a language that is still very recent. In the case of C or C++ this is not the case, as both languages are very mature. Though both are still evolving, the new features keep a high level of compatibility with old code, making the use of those new constructs a programmer's decision. It is not uncommon to establish rules in a project to limit the use of parts of a language (such as RTTI, exceptions, or virtual-functions in inner loops), depending on the proficiency of the programmers or the needs of the project. It is also common for new hardware to support lower level languages first. Due to C being less extensive and lower level than C++, it is easier to check and comply with strict industry guidelines and automate those steps. Another benefit of C is that it is easier for the programmer to do low level optimizations, though most C++ compilers can guarantee near perfect optimizations automatically, a human can still do more and C has less complex structures.

Any of the valid reasons to choose a language over another is mostly due to programmer's choice that indirectly deals with choosing the best tool for the job and having the resources needed to complete it. It would be hard to validate selecting C++ for a project if the available programmers only knew C. Even though in the reverse case it might be expected for a C++ programmer to produce functional C code, the mindset and experience needed are not the same. The same rationale is valid for C programmers and ASM. This is due to the close relations that exist in the language's structure and historic evolution.

One could argue that using the C subset of C++, in a C++ compiler, is the same as using C, but in reality we find that it will generate slightly different results depending on the compiler used.

Java

This is a comparison of the JAVA PROGRAMMING LANGUAGE[85] with the C++ programming language. C++ and Java share many common traits. You can get a better understanding of Java in the JAVA PROGRAMMING WIKIBOOK[86].

Java was created initially to support NETWORK COMPUTING[87] on EMBEDDED SYSTEM[88]s. Java was designed to be extremely PORTABLE[89], SECURE[90], MULTI-THREADED[91] and DISTRIBUTED[92], none of which were design goals for C++. The syntax of Java was chosen to be familiar to C programmers, but direct compatibility with C was not maintained. Java also was specifically designed to be simpler than C++ but it keeps evolving above that simplification.

	C++	**Java**
Compatibility	backwards compatible, including C	backwards compatibility with previous versions
Focus	execution efficiency	developer productivity
Freedom	trusts the programmer	imposes some constraints to the programmer
Memory Management	ARBITRARY MEMORY ACCESS POSSIBLE[93]	memory access only through objects
Code	concise expression	explicit operation
TYPE SAFETY[94]	type casting is restricted greatly	only compatible types can be cast
PROGRAMMING PARADIGM[95]	PROCEDURAL[96] or OBJECT-ORIENTED[97]	object-oriented

85 HTTP://EN.WIKIBOOKS.ORG/WIKI/JAVA%20PROGRAMMING
86 HTTP://EN.WIKIBOOKS.ORG/WIKI/PROGRAMMING%3AJAVA
87 HTTP://EN.WIKIPEDIA.ORG/WIKI/NETWORK%20COMPUTING
88 HTTP://EN.WIKIPEDIA.ORG/WIKI/EMBEDDED%20SYSTEM
89 HTTP://EN.WIKIPEDIA.ORG/WIKI/PORTING
90 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPUTER%20SECURITY
91 HTTP://EN.WIKIPEDIA.ORG/WIKI/THREAD%20%28COMPUTER%20SCIENCE%29
92 HTTP://EN.WIKIPEDIA.ORG/WIKI/DISTRIBUTED%20COMPUTING
93 HTTP://EN.WIKIPEDIA.ORG/WIKI/POINTER
94 HTTP://EN.WIKIPEDIA.ORG/WIKI/TYPE%20SAFETY
95 HTTP://EN.WIKIPEDIA.ORG/WIKI/PROGRAMMING%20PARADIGM
96 HTTP://EN.WIKIPEDIA.ORG/WIKI/PROCEDURAL
97 HTTP://EN.WIKIPEDIA.ORG/WIKI/OBJECT-ORIENTED

	C++	**Java**
Operators	OPERATOR OVERLOADING[98]	meaning of operators immutable
Main Advantage	powerful capabilities of language	feature-rich, easy to use standard library

Differences between C++ and Java are:

- C++ parsing is somewhat more complicated than with Java; for example, `Foo<1>(3);` is a sequence of comparisons if Foo is a variable, but it creates an object if Foo is the name of a class template.
- C++ allows `namespace` level constants, variables, and functions. All such Java declarations must be inside a class or INTERFACE[99].
- CONST[100] in C++ indicates data to be 'read-only,' and is applied to types. `final` in java indicates that the variable is not to be reassigned. For basic types such as `const int` vs `final int` these are identical, but for complex classes, they are different.
- C++ doesn't support constructor delegation.
- C++ runs on the hardware, Java runs on a virtual machine so with C++ you have greater power at the cost of portability.
- C++, `int main()` is a function by itself, without a class.
- C++ access specification (**public**, **private**) is done with labels and in groups.
- C++ access to class members default to `private`, in Java it is package access.
- C++ classes declarations end in a semicolon.
- C++ lacks language level support for garbage collection while Java has built-in garbage collection to handle memory deallocation.
- C++ supports `goto` statements; Java does not, but its LABELED BREAK[101] and LABELED CONTINUE[102] statements provide some structured `goto`-like functionality. In fact, Java enforces STRUCTURED CONTROL FLOW[103], with the goal of code being easier to understand.
- C++ provides some low-level features which Java lacks. In C++, pointers can be used to manipulate specific memory locations, a task necessary for writing

98 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPERATOR%20OVERLOADING
99 HTTP://EN.WIKIPEDIA.ORG/WIKI/INTERFACE%20%28JAVA%29
100 HTTP://EN.WIKIPEDIA.ORG/WIKI/CONST
101 HTTP://EN.WIKIPEDIA.ORG/WIKI/LABELLED%20BREAK
102 HTTP://EN.WIKIPEDIA.ORG/WIKI/LABELLED%20CONTINUE
103 HTTP://EN.WIKIPEDIA.ORG/WIKI/STRUCTURED%20CONTROL%20FLOW

low-level OPERATING SYSTEM[104] components. Similarly, many C++ compilers support INLINE ASSEMBLER[105]. In Java, assembly code can still be accessed as libraries, through the JAVA NATIVE INTERFACE[106]. However, there is significant overhead for each call.

- C++ allows a range of implicit conversions between native types, and also allows the programmer to define implicit conversions involving compound types. However, Java only permits widening conversions between native types to be implicit; any other conversions require explicit cast syntax.
  - A consequence of this is that although loop conditions (`if`, `while` and the exit condition in `for`) in Java and C++ both expect a boolean expression, code such as `if(a = 5)` will cause a compile error in Java because there is no implicit narrowing conversion from int to boolean. This is handy if the code were a typo for `if(a == 5)`, but the need for an explicit cast can add verbosity when statements such as `if (x)` are translated from Java to C++.
- For passing parameters to functions, C++ supports both true PASS-BY-REFERENCE[107] and PASS-BY-VALUE[108]. As in C, the programmer can simulate by-reference parameters with by-value parameters and INDIRECTION[109]. In Java, all parameters are passed by value, but object (non-primitive) parameters are REFERENCE[110] values, meaning INDIRECTION[111] is built-in.
- Generally, Java built-in types are of a specified size and range; whereas C++ types have a variety of possible sizes, ranges and representations, which may even change between different versions of the same compiler, or be configurable via compiler switches.
  - In particular, Java characters are 16-bit UNICODE[112] characters, and strings are composed of a sequence of such characters. C++ offers both narrow and wide characters, but the actual size of each is platform dependent, as is the character set used. Strings can be formed from either type.
- The rounding and precision of floating point values and operations in C++ is platform dependent. Java provides a STRICT FLOATING-POINT MODEL[113] that

104 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPERATING%20SYSTEM
105 HTTP://EN.WIKIPEDIA.ORG/WIKI/INLINE%20ASSEMBLER
106 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20NATIVE%20INTERFACE
107 HTTP://EN.WIKIPEDIA.ORG/WIKI/PASS-BY-REFERENCE
108 HTTP://EN.WIKIPEDIA.ORG/WIKI/PASS-BY-VALUE
109 HTTP://EN.WIKIPEDIA.ORG/WIKI/INDIRECTION
110 HTTP://EN.WIKIPEDIA.ORG/WIKI/REFERENCE%20%28COMPUTER%20SCIENCE%29
111 HTTP://EN.WIKIPEDIA.ORG/WIKI/INDIRECTION
112 HTTP://EN.WIKIPEDIA.ORG/WIKI/UNICODE
113 HTTP://EN.WIKIPEDIA.ORG/WIKI/STRICTFP

guarantees consistent results across platforms, though normally a more lenient mode of operation is used to allow optimal floating-point performance.
- In C++, POINTERS[114] can be manipulated directly as memory address values. Java does not have pointers—it only has object references and array references, neither of which allow direct access to memory addresses. In C++ one can construct pointers to pointers, while Java references only access objects.
- In C++ pointers can point to functions or member functions (FUNCTION POINTER[115]s or FUNCTOR[116]s). The equivalent mechanism in Java uses object or interface references.
- C++ features programmer-defined OPERATOR OVERLOADING[117]. The only overloaded operators in Java are the "+" and "+=" operators, which concatenate strings as well as performing addition.
- Java features standard API[118] support for REFLECTION[119] and DYNAMIC LOADING[120] of arbitrary new code.
- Java has generics. C++ has templates.
- Both Java and C++ distinguish between native types (these are also known as "fundamental" or "built-in" types) and user-defined types (these are also known as "compound" types). In Java, native types have value semantics only, and compound types have reference semantics only. In C++ all types have value semantics, but a reference can be created to any object, which will allow the object to be manipulated via reference semantics.
- C++ supports MULTIPLE INHERITANCE[121] of arbitrary classes. Java supports multiple inheritance of types, but only single inheritance of implementation. In Java, a class can derive from only one class, but a class can implement multiple INTERFACE[122]s.
- Java explicitly distinguishes between interfaces and classes. In C++ multiple inheritance and pure virtual functions makes it possible to define classes that function just as Java interfaces do.

114 http://en.wikipedia.org/wiki/pointers
115 http://en.wikipedia.org/wiki/function%20pointer
116 http://en.wikipedia.org/wiki/functor
117 http://en.wikipedia.org/wiki/operator%20overloading
118 http://en.wikipedia.org/wiki/Application%20programming%20interface
119 http://en.wikipedia.org/wiki/Reflection%20%28computer%20science%29
120 http://en.wikipedia.org/wiki/dynamic%20loading
121 http://en.wikipedia.org/wiki/multiple%20inheritance
122 http://en.wikipedia.org/wiki/Interface%20%28Java%29

- Java has both language and standard library support for MULTI-THREADING[123]. The `synchronized` KEYWORD IN JAVA[124] provides simple and secure MUTEX LOCK[125]s to support multi-threaded applications. While mutex lock mechanisms are available through libraries in C++, the lack of language semantics makes writing THREAD SAFE[126] code more difficult and error prone.

Memory management

- Java requires automatic GARBAGE COLLECTION[127]. Memory management in C++ is usually done by hand, or through SMART POINTER[128]s. The C++ standard permits garbage collection, but does not require it; garbage collection is rarely used in practice. When permitted to relocate objects, modern garbage collectors can improve overall application space and time efficiency over using explicit deallocation.
- C++ can allocate arbitrary blocks of memory. Java only allocates memory through object instantiation. (Note that in Java, the programmer can simulate allocation of arbitrary memory blocks by creating an array of bytes. Still, Java ARRAY[129]s are objects.)
- Java and C++ use different idioms for resource management. Java relies mainly on garbage collection, while C++ relies mainly on the RAII (RESOURCE ACQUISITION IS INITIALIZATION)[130] idiom. This is reflected in several differences between the two languages:
  - In C++ it is common to allocate objects of compound types as local stack-bound variables which are destructed when they go OUT OF SCOPE[131]. In Java compound types are always allocated on the heap and collected by the garbage collector (except in virtual machines that use ESCAPE ANALYSIS[132] to convert heap allocations to stack allocations).

123 HTTP://EN.WIKIPEDIA.ORG/WIKI/MULTI-THREADING
124 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20KEYWORDS
125 HTTP://EN.WIKIPEDIA.ORG/WIKI/MUTUAL%20EXCLUSION
126 HTTP://EN.WIKIPEDIA.ORG/WIKI/THREAD%20SAFE
127 HTTP://EN.WIKIPEDIA.ORG/WIKI/GARBAGE%20COLLECTION%20%28COMPUTER%20SCIENCE%29
128 HTTP://EN.WIKIPEDIA.ORG/WIKI/SMART%20POINTER
129 HTTP://EN.WIKIPEDIA.ORG/WIKI/ARRAY
130 HTTP://EN.WIKIPEDIA.ORG/WIKI/RESOURCE%20ACQUISITION%20IS%20INITIALIZATION
131 Chapter 3.1.9 on page 78
132 HTTP://EN.WIKIPEDIA.ORG/WIKI/ESCAPE%20ANALYSIS

- C++ has destructors, while Java has FINALIZER[133]s. Both are invoked prior to an object's deallocation, but they differ significantly. A C++ object's destructor must be implicitly (in the case of stack-bound variables) or explicitly invoked to deallocate the object. The destructor executes SYNCHRONOUSLY[134] at the point in the program at which the object is deallocated. Synchronous, coordinated uninitialization and deallocation in C++ thus satisfy the RAII idiom. In Java, object deallocation is implicitly handled by the garbage collector. A Java object's finalizer is invoked ASYNCHRONOUSLY[135] some time after it has been accessed for the last time and before it is actually deallocated, which may never happen. Very few objects require finalizers; a finalizer is only required by objects that must guarantee some clean up of the object state prior to deallocation—typically releasing resources external to the JVM. In Java safe synchronous deallocation of resources is performed using the try/finally construct.
- In C++ it is possible to have a DANGLING POINTER[136] – a REFERENCE[137] to an object that has been destructed; attempting to use a dangling pointer typically results in program failure. In Java, the garbage collector won't destruct a referenced object.
- In C++ it is possible to have an object that is allocated, but unreachable. An UNREACHABLE OBJECT[138] is one that has no reachable references to it. An unreachable object cannot be destructed (deallocated), and results in a MEMORY LEAK[139]. By contrast, in Java an object will not be deallocated by the garbage collector *until* it becomes unreachable (by the user program). (Note: WEAK REFERENCE[140]*s* are supported, which work with the Java garbage collector to allow for different *strengths* of reachability.) Garbage collection in Java prevents many memory leaks, but leaks are still possible under some circumstances.

Libraries

133 HTTP://EN.WIKIPEDIA.ORG/WIKI/FINALIZER
134 HTTP://EN.WIKIPEDIA.ORG/WIKI/SYNCHRONIZATION
135 HTTP://EN.WIKIPEDIA.ORG/WIKI/ASYNCHRONY
136 HTTP://EN.WIKIPEDIA.ORG/WIKI/DANGLING%20POINTER
137 HTTP://EN.WIKIPEDIA.ORG/WIKI/REFERENCE%20%28COMPUTER%20SCIENCE%29
138 HTTP://EN.WIKIPEDIA.ORG/WIKI/UNREACHABLE%20OBJECT
139 HTTP://EN.WIKIPEDIA.ORG/WIKI/MEMORY%20LEAK
140 HTTP://EN.WIKIPEDIA.ORG/WIKI/WEAK%20REFERENCE

- C++ STANDARD LIBRARY[141] provides a limited set of basic and relatively general purpose components. Java has a considerably larger standard library. This additional functionality is available for C++ by (often free) third party libraries, but third party libraries do not provide the same ubiquitous cross-platform functionality as standard libraries.
- C++ is mostly BACKWARD COMPATIBLE[142] with C, and C libraries (such as the API[143]s of most OPERATING SYSTEM[144]s) are directly accessible from C++. In Java, the richer functionality its standard library is that it provides CROSS-PLATFORM[145] access to many features typically only available in platform-specific libraries. Direct access from Java to native operating system and hardware functions requires the use of the JAVA NATIVE INTERFACE[146].

Runtime

- C++ is normally compiled directly to MACHINE CODE[147] which is then executed directly by the OPERATING SYSTEM[148]. Java is normally compiled to BYTE-CODE[149] which the JAVA VIRTUAL MACHINE[150] (JVM) then either INTERPRETS[151] or JIT[152] compiles to machine code and then executes.
- Due to the lack of constraints in the use of some C++ language features (e.g. unchecked array access, raw pointers), programming errors can lead to low-level BUFFER OVERFLOW[153]s, PAGE FAULT[154]s, and SEGMENTATION FAULT[155]s. The STANDARD TEMPLATE LIBRARY[156], however, provides higher-level abstractions (like vector, list and map) to help avoid such errors. In Java, such

141 Chapter 5.1.5 on page 499
142 HTTP://EN.WIKIPEDIA.ORG/WIKI/BACKWARD%20COMPATIBLE
143 HTTP://EN.WIKIPEDIA.ORG/WIKI/APPLICATION%20PROGRAMMING%20INTERFACE
144 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPERATING%20SYSTEM
145 HTTP://EN.WIKIPEDIA.ORG/WIKI/CROSS-PLATFORM
146 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20NATIVE%20INTERFACE
147 HTTP://EN.WIKIPEDIA.ORG/WIKI/MACHINE%20CODE
148 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPERATING%20SYSTEM
149 HTTP://EN.WIKIPEDIA.ORG/WIKI/BYTE-CODE
150 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20VIRTUAL%20MACHINE
151 HTTP://EN.WIKIPEDIA.ORG/WIKI/INTERPRETER%20%28COMPUTING%29
152 HTTP://EN.WIKIPEDIA.ORG/WIKI/JUST-IN-TIME%20COMPILATION
153 HTTP://EN.WIKIPEDIA.ORG/WIKI/BUFFER%20OVERFLOW
154 HTTP://EN.WIKIPEDIA.ORG/WIKI/PAGE%20FAULT
155 HTTP://EN.WIKIPEDIA.ORG/WIKI/SEGMENTATION%20FAULT
156 Chapter 5.1.5 on page 499

errors either simply cannot occur or are detected by the JVM[157] and reported to the application in the form of an EXCEPTION[158].
- In Java, BOUNDS CHECKING[159] is implicitly performed for all array access operations. In C++, array access operations on native arrays are not bounds-checked, and bounds checking for random-access element access on standard library collections like std::vector and std::deque is optional.

Miscellaneous

- Java and C++ use different techniques for splitting up code in multiple source files. Java uses a package system that dictates the file name and path for all program definitions. In Java, the compiler imports the executable CLASS FILES[160]. C++ uses a HEADER FILE[161] SOURCE CODE[162] inclusion system for sharing declarations between source files. (See COMPARISON OF IMPORTS AND INCLUDES[163].)
- Templates and macros in C++, including those in the standard library, can result in duplication of similar code after compilation. Second, DYNAMIC LINKING[164] with standard libraries eliminates binding the libraries at compile time.
- C++ compilation features a textual PREPROCESSING[165] phase, while Java does not. Java supports many optimizations that mitigate the need for a preprocessor, but some users add a preprocessing phase to their build process for better support of conditional compilation.
- In Java, arrays are container objects which you can inspect the length of at any time. In both languages, arrays have a fixed size. Further, C++ programmers often refer to an array only by a pointer to its first element, from which they cannot retrieve the array size. However, C++ and Java both provide container classes (**std::vector** and **java.util.ArrayList** respectively) which are re-sizable and store their size.

157 HTTP://EN.WIKIPEDIA.ORG/WIKI/JAVA%20VIRTUAL%20MACHINE
158 HTTP://EN.WIKIPEDIA.ORG/WIKI/EXCEPTION%20HANDLING
159 HTTP://EN.WIKIPEDIA.ORG/WIKI/BOUNDS%20CHECKING
160 HTTP://EN.WIKIPEDIA.ORG/WIKI/CLASS%20%28FILE%20FORMAT%29
161 HTTP://EN.WIKIPEDIA.ORG/WIKI/HEADER%20FILE
162 HTTP://EN.WIKIPEDIA.ORG/WIKI/SOURCE%20CODE
163 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPARISON%20OF%20IMPORTS%20AND%20INCLUDES
164 HTTP://EN.WIKIPEDIA.ORG/WIKI/LIBRARY%20%28COMPUTER%20SCIENCE%29%23DYNAMIC%20LINKING
165 Chapter 3.2.2 on page 98

- Java's division and modulus operators are well defined to truncate to zero. C++ does not specify whether or not these operators truncate to zero or "truncate to -infinity". -3/2 will always be -1 in Java, but a C++ compiler may `return` either -1 or -2, depending on the platform. C99[166] defines division in the same fashion as Java. Both languages guarantee that `(a/b)*b + (a%b) == a` for all a and b (b != 0). The C++ version will sometimes be faster, as it is allowed to pick whichever truncation mode is native to the processor.
- The sizes of integer types is defined in Java (int is 32-bit, long is 64-bit), while in C++ the size of integers and pointers is compiler-dependent. Thus, carefully-written C++ code can take advantage of the 64-bit processor's capabilities while still functioning properly on 32-bit processors. However, C++ programs written without concern for a processor's word size may fail to function properly with some compilers. In contrast, Java's fixed integer sizes mean that programmers need not concern themselves with varying integer sizes, and programs will run exactly the same. This may incur a performance penalty since Java code cannot run using an arbitrary processor's word size.

Performance

Computing performance is a measure of resource consumption when a system of hardware and software performs a piece of computing work such as an algorithm or a transaction. Higher performance is defined to be 'using fewer resources'. Resources of interest include memory, bandwidth, persistent storage and CPU cycles. Because of the high availability of all but the latter on modern desktop and server systems, performance is colloquially taken to mean the least CPU cycles; which often converts directly into the least wall clock time. Comparing the performance of two software languages requires a fixed hardware platform and (often relative) measurements of two or more software subsystems. This section compares the relative computing performance of C++ and Java on common operating systems such as Windows and Linux.

Early versions of Java were significantly outperformed by statically compiled languages such as C++. This is because the program statements of these two closely related languages may compile to a few machine instructions with C++, while compiling into several byte codes involving several machine instructions each when interpreted by a Java JVM[167]. For example:

Java/C++ statement	C++ generated code	Java generated byte code

166 HTTP://EN.WIKIPEDIA.ORG/WIKI/C99
167 HTTP://EN.WIKIPEDIA.ORG/WIKI/JVM

vector[i]++;	mov edx,[ebp+4h]	aload_1
	mov eax,[ebp+1Ch]	iload_2
	inc dword ptr	dup2
	[edx+eax*4]	iaload
		iconst_1
		iadd
		iastore

While this may still be the case for EMBEDDED SYSTEMS[168] because of the requirement for a small footprint, advances in JUST IN TIME (JIT)[169] compiler technology for long-running server and desktop Java processes has closed the performance gap and in some cases given the performance advantage to Java. In effect, Java byte code is compiled into machine instructions at run time, in a similar manner to C++ static compilation, resulting in similar instruction sequences.

C++ is still faster in most operations than Java at the moment, even at low-level and numeric computation. For in-depth information you could check PERFORMANCE OF JAVA VERSUS C++[170]. It's a bit pro-Java but very detailed.

C#

C#[171] (pronounced "See Sharp") is a multi-purpose computer PROGRAMMING LANGUAGE[172] catering to all development needs using MICROSOFT .NET FRAMEWORK[173].

C#'s chief designer was Anders Hejlsberg. Before joining Microsoft in 1996, he worked at Borland developing Turbo Pascal and Delphi. At Microsoft he worked as an architect for J++ and he is still a key participant of the development of the .NET framework.

C# is very similar to Java in that it takes the basic operators and style of C++ but forces programs to be type safe, in that it executes the code in a controlled sandbox called the virtual machine. As such, all code must be encapsulated inside

168 HTTP://EN.WIKIPEDIA.ORG/WIKI/EMBEDDED%20SYSTEMS
169 HTTP://EN.WIKIPEDIA.ORG/WIKI/JUST-IN-TIME%20COMPILATION
170 HTTP://WWW.IDIOM.COM/~{}ZILLA/COMPUTER/JAVACBENCHMARK.HTML
171 HTTP://EN.WIKIBOOKS.ORG/WIKI/SUBJECT%3AC%20SHARP%20PROGRAMMING%20LANGUAGE
172 Chapter 2.1.3 on page 11
173 HTTP://EN.WIKIPEDIA.ORG/WIKI/.NET%20FRAMEWORK

an object, among other things. C# provides many additions to facilitate interaction with MICROSOFT[174]'s Windows, COM, and Visual Basic. C# is a ECMA and ISO standard.

Issues C# vs C++

- Limitation: With C#, features like multiple inheritance from classes (C# implements a different approach called Multiple Implementation, where a class can implement more than one interface), declaring objects on the stack, deterministic destruction (allowing RAII) and allowing default arguments as function parameters (In C# versions < 4.0) will not be available.
- Performance (speed and size): Applications built in C# may not perform as well when compared with native C++. C# has an intrusive garbage collector, reference tracking and other overheads with some of the framework services. The .NET framework alone has a big runtime footprint (˜30 Mb of memory), and requires that several versions of the framework to be installed.
- Flexibility: Due to the dependency on the .NET framework, operating system level functionality (system level APIs) are buffered by a generic set of functions that will reduce some freedoms.
- Runtime Redistribution: Programs need to be distributed with the .NET framework (pre-Windows XP or non Windows Machines), similar to the issue with the Java language, with all the normal upgrade requirements attached.
- Portability: The .NET complete framework is only available on the Windows OS, there is a open-source versions that provides most of the core functionalities, that also supports the GNU-Linux OS, like MONO and Portable.NET HTTP://GETDOTGNU.COM/PNET[175]. There are ECMA and ISO .NET standards for example for C# and the CLI extension to C++.

There are several shortcomings to C++ which are resolved in C#. One of the more subtle ones is the use of reference variables as function arguments. When a code maintainer is looking at C++ source code, if a called function is declared in a header somewhere, the immediate code does not provide any indication that an argument to a function is passed as a reference. An argument passed by reference could be changed after calling the function whereas an argument passed by value cannot be changed. A maintainer not be familiar with the function looking for the location of an unexpected value change of a variable would additionally need to examine the header file for the function in order to determine whether or not that function could have changed the value of the variable. C# insists that the **ref**

174 HTTP://EN.WIKIPEDIA.ORG/WIKI/MICROSOFT
175 HTTP://GETDOTGNU.COM/PNET

keyword be placed in the function call (in addition to the function declaration), thereby cluing the maintainer in that the value could be changed by the function.

Managed C++ (C++/CLI)

Managed C++ is a shorthand notation for Managed Extensions for C++, which are part of the .NET FRAMEWORK[176] from MICROSOFT[177]. This extension of the C++ language was developed to add functionality like automatic garbage collection and heap management, automatic initialization of arrays, and support for multidimensional arrays, simplifying all those details of programming in C++ that would otherwise have to be done by the programmer.

Managed C++ is not compiled to machine code. Rather, it is compiled to COMMON INTERMEDIATE LANGUAGE[178], which is an object-oriented machine language and was formerly known as MSIL.

D

The D programming language, was developed in-house by DIGITAL MARS[179] a small US software company, also known for producing a C compiler (known over time as Datalight C compiler, Zorland C and Zortech C), the first C++ compiler for Windows (originally known as Zortech C++, renamed to Symantec C++, and now Digital Mars C++ (DMC++) and various utilities (such as an IDE[180] for Windows that supports the MFC library).

On their web site, Digital Mars hosts the language specification and a freely-distributable compiler (for Windows and Linux). The compiler back-end is proprietary, only the compiler front-end is licensed under both the Artistic License and the GNU GPL.

Although D originated as a re-engineering of C++ and is predominantly influenced by it, D is not a variant of C++. D has redesigned some C++ features and has been influenced by concepts used in other programming languages, such as Java, C# and Eiffel.

176 http://en.wikipedia.org/wiki/Microsoft%20.NET
177 http://en.wikipedia.org/wiki/Microsoft
178 http://en.wikipedia.org/wiki/Common%20Intermediate%20Language
179 http://en.wikipedia.org/wiki/Digital%20Mars
180 http://en.wikipedia.org/wiki/Integrated%20development%20Environment

Differences between D and C++:

- D does not support multiple inheritance.
- D does not support complex data types with value semantics.

See the D PROGRAMMING[181] book for more details.

2.4 Chapter summary

1. INTRODUCING C++[182]
2. PROGRAMMING LANGUAGES[183]
   a) PROGRAMMING PARADIGMS[184] - the versatility of C++ as a multi-paradigm language, concepts of object-oriented programming (objects and classes, INHERITANCE[185], POLYMORPHISM[186]).
3. COMPARISONS[187] - to other languages, relation to other computer science constructs and idioms.
   a) with C[188]
   b) with JAVA[189]
   c) with C#[190]
   d) with MANAGED C++ (C++/CLI)[191]
   e) with D[192]

1[193] ----

1[194]

181 HTTP://EN.WIKIBOOKS.ORG/WIKI/D%20PROGRAMMING
182 Chapter 2 on page 7
183 Chapter 2.1.3 on page 11
184 Chapter 2.2.3 on page 16
185 Chapter 2.3.4 on page 20
186 Chapter 2.3.4 on page 21
187 Chapter 2.3.6 on page 22
188 Chapter 2.3.7 on page 25
189 Chapter 2.3.7 on page 27
190 Chapter 2.3.7 on page 37
191 Chapter 2.3.7 on page 39
192 Chapter 2.3.7 on page 39
193 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
194 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

3 Fundamentals for getting started

3.1 The code

Code is the string of symbols interpreted by a computer in order to execute a given objective. As with natural languages, code is the result of all the conventions and rules that govern a language. It is what permits implementation of projects in a standard, compilable way. Correctly written code is used to create projects that serve as intermediaries for natural language in order to express meanings and ideas. This, theoretically and actually, allows a computer program to solve any explicitly-defined problem.

undefined behavior

It is also important to note that the language standard leaves some items undefined. In this the C++ language is not alone, but it is at times most vexing to the newcomer, since results may appear inconsistent, especially for the unaware. Of course this becomes most evident when doing cross platform developing requiring the use of different compilers, since the undefined behavior is left to the choices made by each compiler implementor.

Note:
We will try to provide the relevant information as the information is presented, take notice that when we do so we often point you to the documentation of the compiler you are using or note the behavior in the compilers more commonly used.

3.1.1 Programming

The task of programming, while not easy in its execution, is actually fairly simple in its goals. A programmer will envision, or be tasked with, a specific goal. Goals are usually provided in the form of "I want a program that will perform...*fill in the*

blank..." The job of the programmer then is to come up with a "working model" (a model that may consist of one or more ALGORITHMS[1]). That "working model" is sort of an idea of *how* a program will accomplish the goal set out for it. It gives a programmer an idea of what to write in order to turn the idea in to a working program.

Once the programmer has an idea of the structure their program will need to take in order to accomplish the goal, they set about actually writing the program itself, using the selected PROGRAMMING LANGUAGE(S)[2] *keywords*, *functions* and *syntax*. The code that they write is what actually implements the program, or causes it to perform the necessary task, and for that reason, it is sometimes called "implementation code".

3.1.2 What is a program?

To restate the definition, a program is just a sequence of instructions, written in some form of programming language, that tells a computer what to do, and generally how to do it. Everything that a typical user does on a computer is handled and controlled by programs. Programs can contain anything from instructions to solve math problems or send emails, to how to behave when a character is shot in a video game. The computer will follow the instructions of a program one line at a time from the start to the end.

Types of programs

There are all kinds of different programs used today, for all types of purposes. All programs are written with some form of programming language and C++ can be used for in any type of application. Examples of different types of programs, (also called software), include:

Operating Systems

An operating system is responsible for making sure that everything on a computer works the way that it should. It is especially concerned with making certain that your computer's "hardware", (i.e. disk drives, video card and sound card, and etc.) interfaces properly with other programs you have on your computer. Microsoft Windows and Linux are examples of PC operating systems.

1 HTTP://EN.WIKIPEDIA.ORG/WIKI/ALGORITHM

2 Chapter 2.1.3 on page 11

Office Programs

This is a general category for a collection of programs that allow you to compose, view, print or otherwise display different kinds of documents. Often such "suites" come with a word processor for composing letters or reports, a spreadsheet application and a slide-show creator of some kind among other things. Popular examples of Office Suites are Microsoft Office and OpenOffice.org

Web Browsers & Email Clients

A web-browser is a program that allows you to type in an Internet address and then displays that page for you. An email client is a program that allows you to send, receive and compose email messages outside of a web-browser. Often email clients have some capability as a web-browser as well, and some web-browsers have integrated email clients. Well-known web-browsers are Internet Explorer and Firefox, and Email Clients include Microsoft Outlook and Thunderbird. Most are programmed using C++, you can access some as Open-source projects, for instance (HTTP://WWW.MOZILLA.ORG/PROJECTS/FIREFOX/)[3] will help you download and compile Firefox.

Audio/Video Software

These types of software include media players, sound recording software, burning/ripping software, DVD players, etc. Many applications such as Windows Media Player, a popular media player programmed by Microsoft, are examples of audio/video software.

Computer Games

There are countless software titles that are either games or designed to assist with playing games. The category is so wide that it would be impossible to get in to a detailed discussion of all the different kinds of game software without creating a different book! Gaming is one of the most popular activities to engage in on a computer.

Development Software

Development software is software used specifically for programming. It includes software for composing programs in a computer language (sometimes as simple as a text editor like Notepad), for checking to make sure that code is stable and correct (called a debugger), and for compiling that source code into executable programs that can be run later (these are called compilers). Oftentimes, these three separate programs are combined in to one bigger program called an IDE

3 HTTP://WWW.MOZILLA.ORG/PROJECTS/FIREFOX/)

(Integrated Development Environment). There are all kinds of IDEs for every programming language imaginable. A popular C++ IDE for Windows and Linux is the CODE::BLOCKS[4] IDE (FREE AND OPEN SOURCE[5]). The one type of software that you will learn the most about in this book is Development Software.

Types of instructions

As mentioned already, programs are written in many different languages, and for every language, the words and statements used to tell the computer to execute specific commands are different. No matter what words and statements are used though, just about every programming language will include statements that will accomplish the following:

Input

Input is the act of getting information from a keyboard or mouse, or sometimes another program.

Output

Output is the opposite of input; it gives information to the computer monitor or another device or program.

Math/Algorithm

All computer processors (the brain of the computer), have the ability to perform basic mathematical computation, and every programming language has some way of telling it to do so.

Testing

Testing involves telling the computer to check for a certain condition and to do something when that condition is true or false. Conditionals are one of the most important concepts in programming, and all languages have some method of testing conditions.

Repetition

Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of functions that look more or less

4 HTTP://WWW.CODEBLOCKS.ORG/

5 HTTP://WWW.CODEBLOCKS.ORG/FEATURES.SHTML

like these. Thus, one way to describe programming is the process of breaking a large, complex task up into smaller and smaller subtasks until eventually the subtasks are simple enough to be performed with one of these simple functions.

Program execution

Execution starts on MAIN FUNCTION[6], the *entry point* of any (standard-compliant) C++ program. We will cover it when we introduce FUNCTIONS[7].

Execution control or simply *control*, means the process and the location of execution of a program, this has a direct link to PROCEDURAL PROGRAMMING[8]. You will note the mention of *control* as we proceed, as it is necessary concept to explain the order of execution of code and its interpretation by the computer.

Core vs Standard Library

The Core Library consists of the fundamental building blocks of the language itself. Made up of the basic statements that the C++ compiler inherently understands. This includes basic looping constructs such as the if..else, do..while, and for.. statements. The ability to create and modify variables, declare and call functions, and perform basic arithmetic. The Core Library does not include I/O functionality.

The STANDARD LIBRARY[9] is a set of modules that add extended functionality to the language through the use of library or header files. Features such as Input/Output routines, advanced mathematics, and memory allocation functions fall under this heading. All C++ compilers are responsible for providing a Standard Library of functions as outlined by the ANSI/ISO C++ GUIDELINES[10]. More deeper understanding about each module will be provided on the STANDARD C LIBRARY[11], STANDARD INPUT/OUTPUT STREAMS LIBRARY[12] and STANDARD TEMPLATE LIBRARY (STL)[13] sections of this book.

6 Chapter 3.7 on page 229
7 Chapter 3.6.3 on page 229
8 Chapter 2.3.1 on page 16
9 HTTP://EN.WIKIPEDIA.ORG/WIKI/C%2B%2B%20STANDARD%20LIBRARY
10 HTTP://WWW.OPEN-STD.ORG/JTC1/SC22/WG21/
11 Chapter 3.7.10 on page 264
12 Chapter 4.7.3 on page 451
13 Chapter 5.1.5 on page 499

Program organization

How the instructions of a program are written out and stored is generally not a concept determined by a programming language. Punch cards used to be in common use, however under most modern operating systems the instructions are commonly saved as plain text files that can be edited with any text editor. These files are the source of the instructions that make up a program and so are sometimes referred to as **source files** but a more exclusive definition is **source code**.

When referring to **source code** or just **source**, you are considering only the files that contain code, the actual text that makes up the functions (actions) for computer to execute. By referring to **source files** you are extending the idea to not only the files with the instructions that make up the program but all the raw files resources that together can **build** the program. The FILE ORGANIZATION SECTION[14] will cover the different files used in C++ programming and best practices on handling them.

3.1.3 Keywords and identifiers

IDENTIFIERS[15] are names given to variables, functions, objects, etc. to refer to them in the program. C++ identifiers must start with a letter or an underscore character "_", possibly followed by a series of letters, underscores or digits. None of the C++ programming language keywords can be used as identifiers. Identifiers with successive underscores are reserved for use in the header files or by the compiler for special purpose, e.g. name mangling.

Some keywords exists to directly control the compiler's behavior, these keywords are very powerful and must be used with care, they may make a huge difference on the program's compile time and running speed. In the C++ Standard, these keywords are called *Specifiers*.

Special considerations must be given when creating your own identifiers, this will be covered in CODE STYLE CONVENTIONS SECTION[16].

3.1.4 ISO C++ (C++98) keywords

14 Chapter 3.1.5 on page 49
15 HTTP://EN.WIKIPEDIA.ORG/WIKI/IDENTIFIERS
16 Chapter 3.1.8 on page 67

- **and**
- **and_eq**
- **asm**
- **auto**
- **bitand**
- **bitor**
- **bool**
- **break**
- **case**
- **catch**
- **char**
- **CLASS**[17]
- **compl**
- **const**
- `const_cast`
- **continue**
- **default**
- **delete**
- **do**
- **double**
- `dynamic_cast`
- **else**
- `enum`
- **explicit**
- **export**
- `extern`
- **false**
- **float**
- `for`
- **friend**
- `goto`
- **if**
- `inline`
- **int**
- **long**
- **mutable**
- `namespace`
- **new**
- **not**
- **not_eq**
- **operator**
- **or**
- **or_eq**
- **private**
- **protected**
- **public**
- `register`
- `reinterpret_cast`
- `return`
- **short**
- **signed**
- `sizeof`
- `static`
- `static_cast`
- **STRUCT**[18]
- **switch**
- **template**
- **this**
- **throw**
- **true**
- **try**
- `typedef`
- `typeid`
- **typename**
- `union`
- `unsigned`
- **using**
- **virtual**
- **void**
- **volatile**
- **wchar_t**
- **while**
- **xor**
- **xor_eq**

Specific compilers may (in a non-standard compliant mode) also treat some other words as keywords, including **cdecl**, **far**, **fortran**, **huge**, **interrupt**, **near**, **pascal**, **typeof**. Old compilers may recognize the **overload** keyword, an anachronism that has been removed from the language.

The next revision of C++, informally known as C++0x for now, is likely to add some keywords, probably including at least:

- **static_assert**
- **decltype**
- **nullptr**

(These are being considered carefully to minimize breakage to existing code; see HTTP://WWW.OPEN-

17 Chapter 4.2.3 on page 393
18 Chapter 4 on page 385

STD.ORG/JTC1/SC22/WG21/DOCS/PAPERS/2006/N2105.HTML[19] for some details.)

Old compilers may not recognize some or all of the following keywords:

- **and**
- **and_eq**
- **bitand**
- **bitor**
- **bool**
- **catch**
- **compl**
- `const_cast`
- `dynamic_cast`
- **explicit**
- **export**
- **false**
- **mutable**
- `namespace`
- **not**
- **not_eq**
- **or**
- **or_eq**
- `reinterpret_cast`
- `static_cast`
- **template**
- **throw**
- **true**
- **try**
- `typeid`
- **typename**
- **using**
- **wchar_t**
- **xor**
- **xor_eq**

3.1.5 C++ reserved identifiers

Some "nonstandard" identifiers are reserved for distinct uses, to avoid conflicts on the naming of identifiers by vendors, library creators and users in general.

Reserved identifiers include keywords with two consecutive underscores (__), all that start with an underscore followed by an uppercase letter and some other categories of reserved identifiers carried over from the C library specification.

A list of C reserved identifiers can be found at the Internet Wayback Machine archived page: http://web.archive.org/web/20040209031039/http://oakroadsystems.com/tech/c-predef.htm#ReservedIdentifiers

Source code

Source code is the halfway point between human language and machine code. As mentioned before, it can be read by people to an extent, but it can also be parsed (converted) into machine code by a computer. The machine code, represented by

19 HTTP://WWW.OPEN-STD.ORG/JTC1/SC22/WG21/DOCS/PAPERS/2006/N2105.HTML

a series of 1's and 0's, is the only code that the computer can directly understand and act on.

In a small program, you might have as little as a few dozen lines of code at the most, whereas in larger programs, this number might stretch into the thousands or even millions. For this reason, it is sometimes more practical to split large amounts of code across many files. This makes it easier to read, as you can do it bit by bit, and it also reduces compile time of each source file. It takes much less time to compile a lot of small source files than it does to compile a single massive source file.

Managing size is not the only reason to split code, though. Often, especially when a piece of software is being developed by a large team, source code is split. Instead of one massive file, the program is divided into separate files, and each individual file contains the code to perform one particular set of tasks for the overall program. This creates a condition known as *Modularity*. Modularity is a quality that allows source code to be changed, added to, or removed a piece at a time. This has the advantage of allowing many people to work on separate aspects of the same program, thereby allowing it to move faster and more smoothly. Source code for a large project should always be written with modularity in mind. Even when working with small or medium sized projects, it is good to get in the habit of writing code with ease of editing and use in mind.

C++ source code is CASE SENSITIVE[20]. This means that it distinguishes between lowercase and capital letters, so that it sees the words "hello," "Hello," and "HeLlO" as being totally different things. This is important to remember and understand, it will be discussed further in the CODING STYLE CONVENTIONS SECTION[21].

3.1.6 File organization

Most operating systems require files to be designated by a name followed by a specific extension. The C++ standard does not impose any specific rules on how files are named or organized.

The specific conventions for the file organizations has both technical reasons and organizational benefits, very similar to the CODE STYLE CONVENTIONS[22] we will

20 HTTP://EN.WIKIPEDIA.ORG/WIKI/CASE%20SENSITIVITY
21 Chapter 3.1.7 on page 59
22 Chapter 3.1.7 on page 59

examine later. Most of the conventions governing files derive from historical preferences and practices, that are especially related with lower level languages that preceded C++. This is especially true when we take into consideration that C++ was built over the C89 ANSI standard, with compatibility in mind, this has lead to most practices remaining static, except for the operating systems improved support for files and greater ease of management of file resources.

One of the evolutions when dealing with filenames on the language standard was that the default include files would have no extension. Most implementations still provide the old C style headers that use C's file extension ".h" for the C Standard Library, but C++-specific header filenames that were terminated in the same fashion now have no extension (e.g. iostream.h is now iostream). This change to old C++ headers was simultaneous with the implementation of NAMESPACES[23], in particular the std `namespace`.

Note:
Please note that file names and extensions do not include quotes; the quotes were added for clarity in this text.

File names

Selecting a file name shares the same issues to naming variables, functions and in general all things. A name is an identifier that eases not only communication but how things are structured and organized.

Most of the considerations in naming files are commonsensical:

- Names should share the same language: in this, internationalization of the project should be a factor.
- Names should be descriptive, and shared by the related header, the extension will provide the needed distinction.
- Names will be case sensitive, remember to be consistent.

Do not reuse a standard header file name

23 Chapter 3.1.10 on page 79

As you will see later, the C++ Standard defines a LIST OF HEADERS[24]. The behavior is undefined if a file with the same name as a standard header is placed in the search path for included source files.

Extensions

The extension serves one purpose: to indicate to the Operating System, the IDE or the compiler what resides within the file. By itself an extension will not serve as a guarantee for the content.

Since the C language sources usually have the extension ".c" and ".h", in the beginning it was common for C++ source files to share the same extensions or use a distinct variation to clearly indicate the C++ code file. Today this is the practice, most C++ implementation files will use the ".cpp" extension and ".h" for the declaration or header files (the last one is still shared across most assembler and C compilers).

There are other common extensions variations, such as, ".cc", ".C", ".cxx", and ".c++" for "implementation" code. For header files, the same extension variations are used, but the first letter of the extension is usually replaced with an "h" as in, ".hh", ".H", ".hxx", "hpp", ".h++" etc...

Header files will be discussed with more detail later in the PREPROCESSOR SECTION[25] when introducing the #include directive and the standard headers, but in general terms a header file is a special kind of SOURCE CODE[26] file that is included (by the PREPROCESSOR[27]) by way of the #INCLUDE[28] directive, traditionally used at the beginning of a ".cpp" file.

Source code

C++ programs would be compilable even if using a single file, but any complex project will benefit from being split into several source files in order to be manageable and permit re-usability of the code. The beginning programmer sees this as an extra complication, where the benefits are obscure, especially since most of the first attempts will probably result in problems. This section will cover not only the

24 Chapter 3.2.3 on page 100
25 Chapter 3.2.2 on page 98
26 Chapter 3 on page 41
27 Chapter 3.2.2 on page 98
28 Chapter 3.2.3 on page 98

benefits and best practices but also explain how a standardized method will avoid and reduce complexity.

Why split code into several files?

Simple programs will fit into a single source file or at least two, other than that programs can be split across several files in order to:

- Increase organization and better code structure.
- Promote code reuse, on the same project and across projects.
- Facilitate multiple and often simultaneous edits.
- Improve compilation speed.

Source file types

Some authors will refer to files with a .cpp extension as "source files" and files with the .h extension as "header files". However, both of those qualify as source code. As a convention for this book, all code, whether contained within a .cpp extension (where a programmer would put it), or within a .h extension (for headers), will be called source code. Any time we're talking about a .cpp file, we'll call it an "implementation file", and any time we're referring to a header file, we'll call it a "declaration file". You should check the editor/IDE or alter the configuration to a setup that best suits you and others that will read and use this files.

Declaration vs Definition

In general terms a declaration specifies for the linker, the identifier, type and other aspects of language elements such as variables and functions. It is used to announce the existence of the element to the compiler which require variables to be declared before use.

The definition assigns values to an area of memory that was reserved during the declaration phase. For functions, definitions supply the function body. While a variable or function may be declared many times, it is typically defined once.

This is not of much importance for now but is a particular characteristic that impacts how the source code is distributed in files and how it is processed by the

compiler subsystems. It is COVERED IN MORE DETAIL[29] after we introduce you to VARIABLE TYPES[30].

.cpp

An implementation file includes the specific details, that is the definitions, for what is done by the program. While the header file for the light declared what a light could do, the light's .cpp file defines how the light acts.

We will go into much more detail on class definition later; here is a preview:



Figure 4: .cpp files

```
#include "light.h"

Light::Light () : on(false) {
}

void Light::toggle() {
  on = (!on);
}

bool Light::isOn() const {
  return on;
}
```

.h

Header files contain mostly declarations, to be used in the rest of the program. The skeleton of a class is usually provided in a header file, while an accompanying implementation file provides the definitions to put the meat on the bones of it. Header files are not compiled, but rather provided to other parts of the program through the use of `#include`.

29 Chapter 3.3.4 on page 138
30 Chapter 3.3.3 on page 138



Figure 5: .cpp files

A typical header file looks like the following:

```
// Inside sample.h
#ifndef SAMPLE_H
#define SAMPLE_H

// Contents of the header file are placed here.

#endif /* SAMPLE_H */
```

Since header files are included in other files, problems can occur if they are included more than once. This often results in the use of "header guards" using the PREPROCESSOR DIRECTIVES[31] (#ifndef, #define, and #endif). #ifndef checks to see if SAMPLE_H has appeared already, if it has not, the header becomes included and SAMPLE_H is defined. If SAMPLE_H was originally defined, then the file has already been included, and is not included again.



Figure 6: .cpp files

Classes are usually declared inside header files. We will go into much more detail on class declaration later; here is a preview:

```
// Inside light.h
#ifndef LIGHT_H
#define LIGHT_H
```

31 Chapter 3.2.2 on page 98

```
// A light which may be on or off.
class Light {
  private:
    bool on;

  public:
    Light ();       // Makes a new light.
    void toggle (); // If light is on, turn it off, if off, turn it on
    bool isOn();    // Is the light on?
};

#endif /* LIGHT_H - comment indicating which if this goes with */
```

This header file "light.h" declares that there is going to be a light class, and gives the properties of the light, and the methods provided by it. Other programmers can now include this file by typing `#include "light.h"` in their implementation files, which allows them to use this new class. Note how these programmers do not include the actual .cpp file that goes with this class that contains the details of how the light actually works. We'll return to this case study after we discuss implementation files.

Object files

An object file is a temporary file used by the compiler as an intermediate step between the source code and the final executable file.

All other source files that are not or resulted from source code, the support data needed for the build (creation) of the program. The extensions of these files may vary from system to system, since they depend on the IDE/Compiler and necessities of the program, they may include graphic files, or raw data formats.

Object code

The compiler produces machine code equivalent (object code) of the source code, contain the BINARY[32] *language* (*machine language*) instruction to be used by the computer to do as was instructed in the *source code*, that can then be linked into the final program. This step ensures that the code is valid and will sequence into an executable program. Most object files have the file extension (.o) with the same restrictions explained above for the (.cpp/.h) files.

32 HTTP://EN.WIKIPEDIA.ORG/WIKI/BINARY%20AND%20TEXT%20FILES

Libraries

Libraries are commonly distributed in binary form, using the (.lib) extension and header (.h) that provided the interface for its utilization. Libraries can also be dynamically linked and in that case the extension may depend on the target OS, for instance windows libraries as a rule have the (.dll) extension, this will be covered later on in the book in the LIBRARIES SECTION[33] of this book.

Makefiles

It is common for source code to come with a specific script file named "Makefile" (without a standard extension or a standard interpreter). This type of script files is not covered by the C++ Standard, even though it is in common use.

In some projects, especially if dealing with a high level of external dependencies or specific configurations, like supporting special hardware, there is need to automate a vast number of incompatible compile sequences. This scripts are intended to alleviate the task. Explaining in detail the myriad of variations and of possible choices a programmer may make in using (or not) such a system goes beyond the scope of this book. You should check the documentation of the IDE, make tool or the information available on the source you are attempting to compile.

- The APACHE ANT[34] Wikibook describes how to write and use a "build.xml", one way to automate the build process.
- THE "MAKE" WIKIBOOK[35] describes how to write and use a "Makefile", another way to automate the build process.
- ... many IDEs have a "build" button ...

3.1.7 Statements

Most, if not all, programming languages share the concept of a *statement*, also referred to as an *expression*. A statement is a command the programmer gives to the computer.

```
// Example of a single statement
cout << "Hi there!";
```

Each valid C++ statement is terminated by a semicolon (;). The above statement will be examined in detail later on, for now consider that this statement has a

33 Chapter 6.3.3 on page 584
34 HTTP://EN.WIKIBOOKS.ORG/WIKI/APACHE%20ANT
35 HTTP://EN.WIKIBOOKS.ORG/WIKI/MAKE%20

subject (the noun "cout"), a verb ("<<", meaning "output" or "print"), and, in the sense of English grammar, an object (what to print). In this case, the subject "cout" means "the standard console output device", and the verb "<<" means "output the object" — in other words, the command "cout" means "send to the standard output stream," (in this case we assume the default, the console).

The programmer either enters the statement directly to the computer (by typing it while running a special program, called interpreter), or creates a text file with the command in it (you can use any text editor for that), that is latter used with a COMPILER[36]. You could create a file called "hi.txt", put the above command in it, and save that file on the computer.

If one were to write multiple statements, it is recommended that each statement be entered on a separate line.

```
cout << "Hi there!";                  // a statement
cout << "Strange things are afoot..."; // another statement
```

However, there is no problem writing the code this way:

```
cout << "Hi there!"; cout << "Strange things are afoot...";
```

The former code gathers appeal in the developer circles. Writing statements as in the second example only makes your code look more complex and incomprehensible. We will speak of this deeply in the CODING STYLE CONVENTIONS SECTION[37] of the book.

If you have more than one statement in the file, each will be performed in order, top to bottom.

The computer will perform each of these statements sequentially. It is invaluable to be able to "play computer" when programming. Ask yourself, "If I were the computer, what would I do with these statements?" If you're not sure what the answer is, then you are very likely to write incorrect code. Stop and check the language standards and the specific compiler depended implementation if the standard declares it as undefined.

In the above case, the computer will look at the first statement, determine that it is a cout statement, look at what needs to be printed, and display that text on the computer screen. It'll look like this:

```
Hi there!
```

36 Chapter 3.1.10 on page 87
37 Chapter 3.1.7 on page 59

Note that the quotation marks are not there. Their purpose in the program is to tell the computer where the text begins and ends, just like in English prose. The computer will then continue to the next statement, perform its command, and the screen will look like this:

```
Hi there!Strange things are afoot...
```

When the computer gets to the end of the text file, it stops. There are many different kinds of statements, depending on which programming language is being used. For example, there could be a beep statement that causes the computer to output a beep on its speaker, or a window statement that causes a new window to pop up.

Also, the way statements are written will vary depending on the programming language. These differences are fairly superficial. The set of rules like the first two is called a programming language's syntax. The set of verbs is called its library.

```
cout << "Hi there!";
```

Compound statement

Also referred to as *statement blocks* or *code blocks*, consist of one or more statements or commands that are contained between a pair of curly braces `{ }`. Such a block of statements can be named or be provided a condition for execution. Below is how you'd place a series of statements in a block.

```
// Example of a compound statement
{
  int a = 10;
  int b = 20;
  int result = a + b;
}
```

Blocks are used primarily in loops, conditionals and functions. Blocks can be nested inside one another, for instance as an `if` structure inside of a loop inside of a function.

Note:
Statement blocks also create a LOCAL SCOPE[a].

a Chapter 3.1.9 on page 78

Program Control Flow

As seen above the statements are evaluated in the order as they occur (sequentially). The execution of flow begins at the top most statement and proceed downwards till the last statement is encountered. Any single statement can be substituted by a compound statement. There are special statements that can redirect the execution flow based on a condition, those statements are called *branching* statements, described in detail in the CONTROL FLOW CONSTRUCT STATEMENTS SECTION[38] of the book.

3.1.8 Coding style conventions

The use of a guide or set of convention gives programmers a set of rules for code normalization or coding *style* that establishes how to format code, name variables, place comments or any other non language dependent structural decision that is used on the code. This is very important, as you share a project with others. Agreeing to a common set of coding standards and recommendations saves time and effort, by enabling a greater understandings and transparency of the code base, providing a common ground for undocumented structures, making for easy debugging, and increasing code maintainability. These rules may also be referred to as *Source Code Style*, *Code Conventions*, *Coding Standards* or a variation of those.

Many organizations have published C++ style guidelines. A list of different approaches can be found on the C++ CODING CONVENTIONS REFERENCE SECTION[39]. The most commonly used style in C++ programming is ANSI or Allman while much C programming is still done in the Kernighan and Ritchie (K&R) style. You should be warned that this should be one of the first decisions you make on a project and in a democratic environment, a consensus can be very hard to achieve.

Programmers tend to stick to a coding style, they have it automated and any deviation can be very hard to conform with, if you don't have a favorite style try to use the smallest possible variation to a common one or get as broad a view as you can get, permitting you to adapt easily to changes or defend your approach. There is software that can help to format or *beautify* the code, but automation can have its drawbacks. As seen earlier, indentation and the use of white spaces or tabs are completely ignored by the compiler. A coding style should vary depending on the lowest common denominator of the needs to standardize.

Another factor, even if yet to a minimal degree, for the selection of a coding style convention is the IDE (or the code editor) and its capabilities, this can have for

38 Chapter 3.5.2 on page 213
39 Chapter 8.5 on page 653

instance an influence in determining how verbose code should be, the maximum the length of lines, etc. Some editors now have extremely useful features like word completion, refactoring functionalities and other that can make some specifications unnecessary or outright outdated. This will make the adoption of a coding style dependent also on the target code user available software.

Field impacted by the selection of a Code Style are:

- Re-usability
  - Self documenting code
  - Internationalization
  - Maintainability
  - Portability
- Optimization
- Build process
- Error avoidance
- Security

Standardization is important

No matter which particular coding style you pick, once it is selected, it should be kept throughout the same project. Reading code that follows different styles can become very difficult. In the next sections we try to explain why some of the options are common practice without forcing you to adopt a specific style.

Note:
Using a bad Coding Style is worse than having no Coding Style at all, since you will be extending bad practices to all the code base.

25 lines 80 columns

This rule is a commonly recommended, but often countered with argument that the rule is outdated. The rule originates from the time when text-based computer terminals and dot-matrix printers often could display at most 80 columns of text. As such, greater than 80-column text would either inconveniently wrap to the next line, or worse, not display at all.

The physical limitations of the devices asides, this rule often still suggested under the argument that **if you are writing code that will go further than 80 columns or 25 lines, it's time to think about splitting the code into functions**. Smaller

chunks of encapsulated code helps in reviewing the code as it can be seen all at once without scrolling up or down. This modularizes, and thus eases, the programmer mental representation of the project. This practice will save you precious time when you have to return to a project you haven't been working on for 6 months.

For example, you may want to split long output statements across multiple lines:

```
fprintf(stdout,"The quick brown fox jumps over the lazy dog. "
               "The quick brown fox jumps over the lazy dog.\n"
               "The quick brown fox jumps over the lazy dog - %d", 2);
```

This recommended practice relates also to the *0 means success*[40] convention for functions, that we will cover on the FUNCTIONS SECTION[41] of this book.

Whitespace and indentation

Note:
Spaces, tabs and newlines (line breaks) are called *whitespace*. Whitespace is required to separate adjacent words and numbers; they are ignored everywhere else except within quotes and preprocessor directives

Conventions followed when using whitespace to improve the readability of code is called an **indentation style**. Every block of code and every definition should follow a consistent indention style. This usually means everything within `{` and `}`. However, the same thing goes for one-line code blocks.

Use a fixed number of spaces for indentation. Recommendations vary; 2, 3, 4, 8 are all common numbers. If you use tabs for indention you have to be aware that editors and printers may deal with, and expand, tabs differently. The K&R standard recommends an indentation size of 4 spaces.

The use of tab is controversial, the basic premise is that it reduces source code portability, since the same source code loaded into different editors with distinct setting will not look alike. This is one of the primary reasons why some programmers prefer the consistency of using spaces (or configure the editor to replace the use of the tab key with the necessary number of spaces).

For example, a program could as well be written using as follows:

40 Chapter 3.7 on page 229
41 Chapter 3.6.3 on page 229

```
// Using an indentation size of 2
if ( a > 5 )  { b=a; a++; }
```

However, the same code could be made much more readable with proper indentation:

```
// Using an indentation size of 2
if ( a > 5 )  {
  b = a;
  a++;
}

// Using an indentation size of 4
if ( a > 5 )
{
    b = a;
    a++;
}
```

Placement of braces (CURLY BRACKETS[42])

As we have seen early on the STATEMENTS SECTION[43], *compound statement* are very important in C++, they also are subject of different coding styles, that recommend different placements of opening and closing braces ({ and }). Some recommend putting the opening brace on the line with the statement, at the end (K&R[44]). Others recommend putting these on a line by itself, but not indented (ANSI C++). GNU recommends putting braces on a line by itself, and indenting them half-way. We recommend picking one brace-placement style and sticking with it.

Examples:

```
if (a > 5) {
  // This is K&R style
}

if (a > 5)
{
  // This is ANSI C++ style
}

if (a > 5)
  {
```

42 HTTP://EN.WIKIPEDIA.ORG/WIKI/CURLY%20BRACKET%20PROGRAMMING%20LANGUAGE

43 Chapter 3.1.6 on page 56

44 HTTP://EN.WIKIPEDIA.ORG/WIKI/THE%20C%20PROGRAMMING%20LANGUAGE%20%28BOOK%29

```
    // This is GNU style
  }
```

Comments

Comments are portions of the code ignored by the compiler which allow the user to make simple notes in the relevant areas of the source code. Comments come either in block form or as single lines.

- **Single-line comments** (informally, *C++ style*), start with `//` and continue until the end of the line. If the last character in a comment line is a `\` the comment will continue in the next line.
- **Multi-line comments** (informally, *C style*), start with `/*` and end with `*/`.

> **Note:**
> Since the 1999 revision, C also allows *C++ style* comments, so the informal names are largely of historical interest that serves to make a distinction of the two methods of commenting.

We will now describe how a comment can be added to the source code, but not where, how, and when to comment; we will get into that later.

C style comments

If you use this kind of comment try to use it like this... Commented

```
/*void EventLoop(); /**/
```

or for multiple lines

```
/*
void EventLoop();
void EventLoop();
/**/
```

this opens you the option to do this... Uncommented

```
void EventLoop(); /**/
```

or for multiple lines

```
void EventLoop();
void EventLoop();
/**/
```

Note:
Some compilers may generate errors/warnings.

Try to avoid using C style inside a function because of the non nesting facility of C style (most editors now have some sort of coloring ability that prevents this kind of error, but it was very common to miss it, and you shouldn't make assumptions on how the code is read).

... by removing only the start of comment and so activating the next one, you did re-activate the commented code, because if you start a comment this way it will be valid until it finds the close of comment `*/`.

Note:
Remember that C-style comments `/* like this */` do not "nest", i.e., you can't write

```
int function() /* This is a comment /* {   return 0;  } and this is
the same comment */ so this isn't in the comment, and will give an error*/
```

because of the text `so this is not in the comment */` at the end of the line, which is not inside the comment; the comment ends at the first `*/` sequence it finds, ignoring any interim `/*` sequence, which might look to human readers like the start of a nested comment.

C++ style **comments**
Examples:

```
// This is a single one line comment
```

or

```
if (expression) // This needs a comment
{
  statements;
}
else
{
  statements;
}
```

The backslash is a continuation character and will continue the comment to the following line:

```
// This comment will also comment the following line \

std::cout << "This line will not print" << std::endl;
```

Using comments to temporarily ignore code

Comments are also sometimes used to enclose code that we temporarily want the compiler to ignore. This can be useful in finding errors in the program. If a program does not give the desired result, it might be possible to track which particular statement contains the error by commenting out code.

Example with *C style* comments

```
/* This is a single line comment */
```

or

```
/*
   This is a multiple line comment
*/
```

C* and *C++ style

Combining multi-line comments (`/* */`) with c++ comments (`//`) to comment out multiple lines of code:

Commenting out the code:

```
/*
void EventLoop();
void EventLoop();
void EventLoop();
void EventLoop();
void EventLoop();
//*/
```

uncommenting the code chunk

```
//*
void EventLoop();
void EventLoop();
void EventLoop();
void EventLoop();
```

```
void EventLoop();
//*/
```

This works because a `//*` is still a c++ comment. And `//*/` acts as a c++ comment and a multi-line comment terminator. However this doesn't work if there are any multi-line comments are used for function descriptions.

Note on doing it with preprocessor statements

Another way (considered bad practice) is to selectively enable disable sections of code:

```
#if(0)   // Change this to 1 to uncomments.
void EventLoop();
#endif
```

this is considered a bad practice because the code often becomes illegible when several #if's are mixed, if you use them don't forget to add a comment at the #endif saying what #if it correspond

```
#if (FEATURE_1 == 1)
do_something;
#endif //FEATURE_1 == 1
```

you can prevent illegibility by using `inline` functions (often considered better than macros for legibility with no performance cost) containing only 2 sections in #if #else #endif

```
inline do_test()
  {
    #if (Feature_1 == 1)
      do_something
    #endif  //FEATURE_1 == 1
  }
```

and call

```
do_test();
```

in the program

Note:
The use of one-line C-style comments should be avoided as they are considered outdated. Mixing C and C++ style single-line comments is considered poor practice. One exception, that is commonly used, is to disable a specific part of code in the middle of a single line statement for test/debug purposes, in release code any need for such action should be removed.

45

Naming identifiers

C++'s restriction about the names of IDENTIFIERS[46] and ITS *keywords*[47] have already been covered, on the CODE SECTION[48]. They leave a lot of freedom in naming, one could use specific prefixes or suffixes, start names with an initial upper or lower case letter, keep all the letters in a single case or, with compound words, use a word separator character like "_" or flip the case of the first letter of each component word.

Note:
It is also important to remember to avoid collisions with the OS's APIs (depending on the portability requirements) or other standards. For instance POSIX's keywords terminate in "_t".

Hungarian notation

Hungarian notation, now also referred to as Apps Hungarian, was invented by Charles Simonyi (a programmer who worked at Xerox PARC circa 1972-1981, and who later became Chief Architect at Microsoft); and has been until recently the preeminent naming convention used in most Microsoft code. It uses prefixes (like "m_" to indicate member variables and "p" to indicate pointers), while the rest of the identifier is normally written out using some form of mixed capitals. We mention this convention because you will very probably find it in use, even

45 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
46 HTTP://EN.WIKIPEDIA.ORG/WIKI/IDENTIFIERS
47 Chapter 3.1.3 on page 46
48 Chapter 3 on page 41

more probable if you do any programming in Windows, if you are interested on learning more you can check WIKIPEDIA'S ENTRY ON THIS NOTATION[49].

This notation is considered outdated, since it is highly prone to errors and requires some effort to maintain without any real benefit in today's IDEs. Today refactoring is an everyday task, the IDEs have evolved to provide help with identifier pop-ups and the use of color schemes. All these informational aids reduce the need for this notation.

Leading underscores
In most contexts, leading underscores are better avoided. They are reserved for the compiler or internal variables of a library, and can make your code less portable and more difficult to maintain. Those variables can also be stripped from a library (i.e. the variable is not accessible anymore, it is hidden from external world) so unless you want to override an internal variable of a library, do not do it.

Reusing existing names
Do not use the names of standard library functions and objects for your identifiers as these names are considered reserved words and programs may become difficult to understand when used in unexpected ways.

Sensible names
Always use good, unabbreviated, correctly-spelled meaningful names.

Prefer the English language (since C++ and most libraries already use English) and avoid short cryptic names. This will make it easier to read and to type a name without having to look it up.

49 HTTP://EN.WIKIPEDIA.ORG/WIKI/HUNGARIAN%20NOTATION

Note:
It is acceptable to ignore this rule for loop variables and variables used within a small scope (˜20 lines), they may be given short names to save space if the purpose of that variable is obvious enough. Historically the most commonly used variable name in this cases is "i".
The "i" may derive from the word "increment" or "index". The "i" is very commonly found in `for` loops that does fit nicely the specification for the use of such variable names.
In early Fortran compilers, the letters i through q represented integer variables - and by convention the first few (i, j, k) were often used as loop counters.

Names indicate purpose
An identifier should indicate the function of the variable/function/etc. that it represents, e.g. `foobar` is probably not a good name for a variable storing the age of a person.

Identifier names should also be descriptive. `n` might not be a good name for a global variable representing the number of employees. However, a good medium between long names and lots of typing has to be found. Therefore, this rule can be relaxed for variables that are used in a SMALL SCOPE OR CONTEXT[50]. Many programmers prefer short variables (such as i) as loop iterators.

Capitalization
Conventionally, variable names start with a lower case character. In identifiers which contain more than one natural language words, either underscores or capitalization is used to delimit the words, e.g. `num_chars` (K&R style) or `numChars` (Java style). It is recommended that you pick one notation and do not mix them within one project.

Constants
When naming #defines, constant variables, `enum` constants. and macros put in all uppercase using '_' separators; this makes it very clear that the value is not alterable and in the case of macros, makes it clear that you are using a construct that requires care.

50 Chapter 3.1.9 on page 78

Note:
There is a large school of thought that names LIKE_THIS should be used *only* for macros, so that the name space used for macros (which do not respect C++ scopes) does not overlap with the name space used for other identifiers. As is usual in C++ naming conventions, there is not a single universally agreed standard. The most important thing is usually to be consistent.

Functions and member functions

The name given to functions and member functions should be descriptive and make it clear what it does. Since usually functions and member functions perform actions, the best name choices typically contain a mix of verbs and nouns in them such as CheckForErrors() instead of ErrorCheck() and dump_data_to_file() instead of data_file(). Clear and descriptive names for functions and member functions can sometimes make guessing correctly what functions and member functions do easier, aiding in making code more self documenting. By following this and other naming conventions programs can be read more naturally.

People seem to have very different intuitions when using names containing abbreviations. It is best to settle on one strategy so the names are absolutely predictable. Take for example NetworkABCKey. Notice how the C from ABC and K from key are confused. Some people do not mind this and others just hate it so you'll find different policies in different code so you never know what to call something.

Prefixes and suffixes are sometimes useful:

- **Min** - to mean the minimum value something can have.
- **Max** - to mean the maximum value something can have.
- **Cnt** - the current count of something.
- **Count** - the current count of something.
- **Num** - the current number of something.
- **Key** - key value.
- **Hash** - hash value.
- **Size** - the current size of something.
- **Len** - the current length of something.
- **Pos** - the current position of something.
- **Limit** - the current limit of something.
- **Is** - asking if something is true.
- **Not** - asking if something is not true.
- **Has** - asking if something has a specific value, attribute or property.
- **Can** - asking if something can be done.

- **Get** - get a value.
- **Set** - set a value.

Examples

In most contexts, leading underscores are also better avoided. For example, these are valid identifiers:

- **i** *loop value*
- **numberOfCharacters** *number of characters*
- **number_of_chars** *number of characters*
- **num_chars** *number of characters*
- **get_number_of_characters()** *get the number of characters*
- **get_number_of_chars()** *get the number of characters*
- **is_character_limit()** *is this the character limit?*
- **is_char_limit()** *is this the character limit?*
- **character_max()** *maximum number of a character*
- **charMax()** *maximum number of a character*
- **CharMin()** *minimum number of a character*

These are also valid identifiers but can you tell what they mean?:

- **num1**
- **do_this()**
- **g()**
- **hxq**

The following are valid identifiers but better avoided:

- **_num** as it could be used by the compiler/system headers
- **num__chars** *as it could be used by the compiler/system headers*
- **main** *as there is potential for confusion*
- **cout** *as there is potential for confusion*

The following are not valid identifiers:

- **if** *as it is a keyword*
- **4nums** *as it starts with a digit*
- **number of characters** *as spaces are not allowed within an identifier*

Explicitness or implicitness

This can be defended both ways. If defaulting to implicitness, this means less typing but also may create wrong assumptions on the human reader and for the

compiler (depending on the situation) to do extra work, on the other hand if you write more keywords and are explicit on your intentions the resulting code will be clearer and reduces errors (enabling hidden errors to be found), or more defined (self documented) but this may also lead to added limitations to the code's evolution (like we will see with the use of const). This is a thin line were an equilibrium must be reached in accord to the projects nature, and the available capabilities of the editor, code completion, syntax coloring and hovering tooltips reduces much of the work. The important fact is to be consistent as with any other rule.

`inline`
The choice of using of `inline` even if the member function is implicitly inlined.

const
Unless you plan on modifying it, you're arguably better off using const data types. The compiler can easily optimize more with this restriction, and you're unlikely to accidentally corrupt the data. Ensure that your methods take const data types unless you absolutely have to modify the parameters. Similarly, when implementing accessors for private member data, you should in most cases `return` a const. This will ensure that if the object that you're operating on is passed as const, methods that do not affect the data stored in the object still work as they should and can be called. For example, for an object containing a person, a getName() should `return` a const data type where as walk() might be non-const as it might change some internal data in the Person such as tiredness.

`typedef`
It is common practice to avoid using the `typedef` keyword since it can obfuscate code if not properly used or it can cause programmers to accidentally misuse large structures thinking them to be simple types. If used, define a set of rules for the types you rename and be sure to document them.

volatile
This keyword informs the compiler that the variable it is qualifying as volatile (can change at anytime) is excluded from any optimization techniques. Usage of this variable should be reserved for variables that are known to be modified due to an external influence of a program (whether it's hardware update, third party application, or another thread in the application).

Since the volatile keyword impacts performance, you should consider a different design that avoids this situation: most platforms where this keyword is necessary provide an alternative that helps maintain scalable performance.

Note that using volatile was not intended to be used as a threading or synchronization primitive, nor are operations on a volatile variable guaranteed to be atomic.

Pointer declaration

Due to historical reasons some programmers refer to a specific use as:

```
// C code style
int *z;

// C++ code style
int* z;
```

The second variation is by far the preferred by C++ programmers and will help identify a C programmer or legacy code.

One argument against the C++ code style version is when chaining declarations of more than one item, like:

```
// C code style
int *ptrA, *ptrB;

// C++ code style
int* ptrC, ptrD;
```

As you can see, in this case, the C code style makes it more obvious that ptrA and ptrB are pointers to int, and the C++ code style makes it less obvious that ptrD is an int, not a pointer to int.

It is rare to use chains of multiple objects in C++ code with the exception of the basic types and even so it is a not often used and it is extremely rare to see it used in pointers or other complex types, since it will make it harder to for a human to visually parse the code.

```
// C++ code style
int* ptrC;
int D;
```

References

3.1.9 Document your code

There are a number of good reasons to document your code, and a number of aspects of it that can be documented. Documentation provides you with a shortcut for obtaining an overview of the system or for understanding the code that provides a particular feature.

"Good code is its own best documentation."
—Steve McConnell

Why?

The purpose of comments is to explain and clarify the source code to anyone examining it (or just as a reminder to yourself). Good commenting conventions are essential to any non-trivial program so that a person reading the code can understand what it is expected to do and to make it easy to follow on the rest of the code. In the next topics some of the most **How?** and **When?** rules to use comments will be listed for you.

Documentation of programming is essential when programming not just in C++, but in any programming language. Many companies have moved away from the idea of "hero programmers" (i.e., one programmer who codes for the entire company) to a concept of groups of programmers working in a team. Many times programmers will only be working on small parts of a larger project. In this particular case, documentation is essential because:

- Other programmers may be tasked to develop your project;
- Your finished project may be submitted to editors to assemble your code into other projects;
- A person other than you may be required to read, understand, and present your code.

Even if you are not programming for a living or for a company, documentation of your code is still essential. Though many programs can be completed in a few hours, more complex programs can take longer time to complete (days, weeks, etc.). In this case, documentation is essential because:

- You may not be able to work on your project in one session;
- It provides a reference to what was changed the last time you programmed;

- It allows you to record *why* you made the decisions you did, including why you chose **not** to explore certain solutions;
- It can provide a place to document known limitations and bugs (for the latter a defect tracking system may be the appropriate place for documentation);
- It allows easy searching and referencing within the program (from a non-technical stance);
- It is considered to be good programming practice.

For the appropriate audience

Comments should be written for the appropriate audience. When writing code to be read by those who are in the initial stages of learning a new programming language, it can be helpful to include a lot of comments about what the code does. For "production" code, written to be read by professionals, it is considered unhelpful and counterproductive to include comments which say things that are already clear in the code. Some from the EXTREME PROGRAMMING[51] community say that excessive commenting is indicative of CODE SMELL[52] -- which is *not* to say that comments are bad, but that they are often a clue that code would benefit from REFACTORING[53]. Adding comments as an alternative to writing understandable code is considered poor practice.

What?

What needs to be documented in a program/source code can be divided into what is documented before the specific program execution (that is before "main") and what is executed ("what is in main").

Documentation before program execution:

- Programmer information and license information (if applicable)
- User defined function declarations
- Interfaces
- Context
- Relevant standards/specifications
- Algorithm steps
- How to convert the source code into executable file(s) (perhaps by using MAKE[54])

51 HTTP://EN.WIKIPEDIA.ORG/WIKI/EXTREME%20PROGRAMMING
52 HTTP://EN.WIKIPEDIA.ORG/WIKI/CODE%20SMELL
53 HTTP://EN.WIKIPEDIA.ORG/WIKI/REFACTORING
54 HTTP://EN.WIKIBOOKS.ORG/WIKI/MAKE

Documentation for code inside main:

- Statements, Loops, and Cases
- Public and Private Sectors within Classes
- Algorithms used
- Unusual features of the implementation
- Reasons why other choices have been avoided
- User defined function implementation

If used carelessly comments can make source code hard to read and maintain and may be even unnecessary if the code is self-explanatory -- but remember that what seems self-explanatory today may not seem the same six months or six years from now.

Document decisions

Comments should document decisions. At every point where you had a choice of what to do place a comment describing which choice you made and why. Archaeologists will find this the most useful information.

Comment layout

Each part of the project should at least have a single comment layout, and it would be better yet to have the complete project share the same layout if possible.

How?

Documentation can be done within the source code itself through the use of comments (as seen above) in a language understandable to the intended audience. It is good practice to do it in English as the C++ language is itself English based and English being the current LINGUA FRANCA[55] of international business, science, technology and aviation, you will ensure support for the broadest audience possible.

Comments are useful in documenting portions of an algorithm to be executed, explaining function calls and variable names, or providing reasons as to why a specific choice or method was used. Block comments are used as follows:

```
/*
```

55 HTTP://EN.WIKIPEDIA.ORG/WIKI/LINGUA%20FRANCA

```
get timepunch algorithm - this algorithm gets a time punch for use later
1. user enters their number and selects "in" or "out"
2. time is retrieved from the computer
3. time punch is assigned to user
*/
```

Alternately, line comments can be used as follows:

```
GetPunch(user_id, time, punch); //this function gets the time punch
```

An example of a full program using comments as documentation is:

```
/*
Chris Seedyk
BORD Technologies
29 December 2006
Test
*/
int main()
{
 cout << "Hello world!" << endl; //predefined cout prints stuff in " " to screen
 return 0;
}
```

It should be noted that while comments are useful for in-program documentation, it is also a good idea to have an external form of documentation separate from the source code as well, but remember to think first on how the source will be distributed before making references to external information on the code comments.

Commenting code is also no substitute for well-planned and meaningful variable, function, and class names. This is often called "self-documenting code," as it is easy to see from a carefully chosen and descriptive name what the variable, function, or class is meant to do. To illustrate this point, note the relatively equal simplicity with which the following two ways of documenting code, despite the use of comments in the first and their absence in the second, are understood. The first style is often encountered in very old C source by people who understood well what they were doing and had no doubt anyone else might not comprehend it. The second style is more "human-friendly" and while much easier to read is nevertheless not as frequently encountered.

```
// Returns the area of a triangle cast as an int
int area_ftoi(float a, float b) { return (int) a * b / 2; }

int iTriangleArea(float fBase, float fHeight)
{
   return (int) fBase * fHeight / 2;
}
```

Both functions perform the same task, however the second has such practical names chosen for the function and the variables that its purpose is clear even without comments. As the complexity of the code increases, well-chosen naming schemes increase vastly in importance.

Regardless of what method is preferred, comments in code are helpful, save time (and headaches), and ensure that both the author and others understand the layout and purpose of the program fully.

Automatic documentation

Various tools are available to help with documenting C++ code; LITERATE PROGRAMMING[56] is a whole school of thought on how to approach this, but a very effective tool is DOXYGEN[57] (also supports several languages), it can even use hand written comments in order to generate more than the bare structure of the code, bringing Javadoc-like documentation comments to C++ and can generate documentation in HTML, PDF and other formats.

Comments should tell a story

Consider your comments a story describing the system. Expect your comments to be extracted by a robot and formed into a manual page. Class comments are one part of the story, method signature comments are another part of the story, method arguments another part, and method implementation yet another part. All these parts should weave together and inform someone else at another point of time just exactly what you did and why.

Do not use comments for flowcharts or pseudo-code

You should refrain from using comments to do ASCII art or pseudo-code (some programmers attempt to explain their code with an ASCII-art flowchart). If you want to flowchart or otherwise model your design there are tools that will do a better job at it using standardized methods. See for example: UML[58].

56 HTTP://EN.WIKIPEDIA.ORG/WIKI/LITERATE%20PROGRAMMING

57 HTTP://WWW.DOXYGEN.ORG

58 HTTP://EN.WIKIPEDIA.ORG/WIKI/UNIFIED%20MODELING%20LANGUAGE

3.1.10 Scope

In any language, **scope** (the context; what is the background) has a high impact on a given action or statement validity. The same is true in a programming language.

In a program we may have various constructs, may they be objects, variables or any other such. They come into existence from the point where you declare them (before they are declared they are unknown) and then, at some point, they are destroyed (as we will see there are many reasons to be so) and all are destroyed when your program terminates.

We will see THAT VARIABLES HAVE A FINITE LIFE-TIME WHEN YOUR PROGRAM EXECUTES[59], that the scope of an object or variable is simply that part of a program in which the variable name exists or is visible to the compiler.

Global scope

The default scope is defined as **global scope**, this is commonly used to define and use global variables or other global constructs (classes, structure, functions, etc...), this makes them valid and visible to the compiler at all times.

Note:
It is considered a good practice, if possible and as a way to reduce complexity and name collisions, to use a `namespace` scope for hiding the otherwise global elements, without removing their validity.

Local scope

A **local scope** relates to the scope created inside a COMPOUND STATEMENT[60].

Note:
The only exceptional case is the `for` keyword. In that case the variables declared on the `for` *initialization* section will be part of the local scope.

59 Chapter 3.3 on page 121
60 Chapter 3.1.7 on page 58

`namespace`

The **namespace** keyword allows you to create a new scope. The name is optional, and can be omitted to create an *unnamed namespace*. Once you create a `namespace`, you'll have to refer to it explicitly or use the `using` keyword. A namespace is defined with a `namespace` block.

Syntax

```
namespace name {
declaration-list;
}
```

In many PROGRAMMING LANGUAGE[61]s, a NAMESPACE[62] is a context for IDENTIFIER[63]s. C++ can handle multiple namespaces within the language. By using `namespace` (or the `using namespace` keyword), one is offered a clean way to aggregate code under a shared *label*, so as to prevent naming collisions or just to ease recall and use of very specific scopes. There are other "name spaces" besides "namespaces"; this can be confusing.

Name spaces (note the space there), as we will see, go beyond the concept of scope by providing an easy way to differentiate what is being called/used. As we will see, classes are also name spaces, but they are not namespaces.

Note:
Use `namespace` only for convenience or real need, like aggregation of related code, do not use it in a way to make code overcomplicated for you and others

Example

```
namespace foo {
  int bar;
}
```

61 HTTP://EN.WIKIPEDIA.ORG/WIKI/PROGRAMMING%20LANGUAGE
62 HTTP://EN.WIKIPEDIA.ORG/WIKI/NAMESPACE%20%28COMPUTER%20SCIENCE%29
63 HTTP://EN.WIKIPEDIA.ORG/WIKI/IDENTIFIER

Within this block, identifiers can be used exactly as they are declared. Outside of this block, the `namespace` specifier must be prefixed (that is, it must be *qualified*). For example, outside of `namespace foo`, `bar` must be written `foo::bar`.

C++ includes another construct which makes this verbosity unnecessary. By adding the line `using namespace foo;` to a piece of code, the prefix `foo::` is no longer needed.

unnamed namespace

A `namespace` without a name is called an **unnamed namespace**. For such a `namespace`, a unique name will be generated for each translation unit. It is not possible to apply the `using` keyword to *unnamed namespaces*, so an *unnamed namespace* works as if the using keyword has been applied to it.

Syntax

```
namespace {
declaration-list;
}
```

namespace alias

You can create new names (aliases) for namespaces, including *nested namespaces*.

Syntax

```
namespace identifier = namespace-specifier;
```

using namespaces

using

```
using namespace std;
```

This `using`-directive indicates that any names used but not declared within the program should be sought in the 'standard (std)' `namespace`.

Note:
It is always a bad idea to use a `using` directive in a header file, as it affects every use of that header file and would make difficult its use in other derived projects; there is no way to "undo" or restrict the use of that directive. Also don't use it before an `#include` directive.

To make a single name from a `namespace` available, the following `using`-declaration exists:

```
using foo::bar;
```

After this declaration, the name `bar` can be used inside the current `namespace` instead of the more verbose version `foo::bar`. Note that programmers often use the terms declaration and directive interchangeably, despite their technically different meanings.

It is good practice to use the narrow second form (using declaration), because the broad first form (using directive) might make more names available than desired. Example:

```
namespace foo {
  int bar;
  double pi;
}

using namespace foo;

int* pi;
pi = &bar;  // ambiguity: pi or foo::pi?
```

In that case the declaration `using foo::bar;` would have made only `foo::bar` available, avoiding the clash of `pi` and `foo::pi`. This problem (the collision of identically-named variables or functions) is called "namespace pollution" and as a rule should be avoided wherever possible.

`using`-declarations can appear in a lot of different places. Among them are:

- namespaces (including the default namespace)
- functions

A `using`-declaration makes the name (or `namespace`) available in the scope of the declaration. Example:

```
namespace foo {
  namespace bar {
   double pi;
```

```
  }

  using bar::pi;
  // bar::pi can be abbreviated as pi
}

// here, pi is no longer an abbreviation. Instead, foo::bar::pi must be used.
```

Namespaces are hierarchical. Within the hypothetical `namespace food::fruit`, the identifier `orange` refers to `food::fruit::orange` if it exists, or if not, then `food::orange` if that exists. If neither exist, `orange` refers to an identifier in the default `namespace`.

Code that is not explicitly declared within a `namespace` is considered to be in the *default namespace*.

Another property of namespaces is that they are *open*. Once a `namespace` is declared, it can be redeclared (*reopened*) and `namespace` members can be added. Example:

```
namespace foo {
  int bar;
}

// ...

namespace foo {
  double pi;
}
```

Namespaces are most often used to avoid naming collisions. Although namespaces are used extensively in recent C++ code, most older code does not use this facility. For example, the entire standard library is defined within `namespace std`, and in earlier standards of the language, in the *default namespace*.

For a long *namespace name*, a shorter alias can be defined (a `namespace` *alias* declaration). Example:

```
namespace ultra_cool_library_for_image_processing_version_1_0 {
  int foo;
}

namespace improc1 = ultra_cool_library_for_image_processing_version_1_0;
// from here, the above foo can be accessed as improc1::foo
```

There exists a special `namespace`: the *unnamed namespace*. This `namespace` is used for names which are private to a particular source file or other `namespace`:

```
namespace {
  int some_private_variable;
```

```
}
// can use some_private_variable here
```

In the surrounding scope, members of an *unnamed namespace* can be accessed without qualifying, i.e. without prefixing with the namespace name and `::` (since the namespace doesn't have a name). If the surrounding scope is a `namespace`, members can be treated and accessed as a member of it. However, if the surrounding scope is a file, members cannot be accessed from any other source file, as there is no way to name the file as a scope. An *unnamed namespace* declaration is semantically equivalent to the following construct

```
namespace $$$ {
  // ...
}
using namespace $$$;
```

where `$$$` is a unique identifier manufactured by the compiler.

As you can nest an *unnamed namespace* in an ordinary `namespace`, and vice versa, you can also nest two unnamed namespaces.

```
namespace {

  namespace {
    // ok
  }

}
```

Note:
If you enable the use of a `namespace` in the code, all the code will use it (you can't define sections that will and exclude others), you can however use *nested namespace* declarations to restrict its scope.

Because of space considerations, we cannot actually show the `namespace` command being used properly: it would require a very large program to show it working usefully. However, we can illustrate the concept itself easily.

```
// Namespaces Program, an example to illustrate the use of namespaces
#include <iostream>

namespace first {
  int first1;
  int x;
}

namespace second {
  int second1;
```

```
  int x;
}

namespace first {
  int first2;
}

int main(){
  //first1 = 1;
  first::first1 = 1;
  using namespace first;
  first1 = 1;
  x = 1;
  second::x = 1;
  using namespace second;

  //x = 1;
  first::x = 1;
  second::x = 1;
  first2 = 1;

  //cout << 'X';
  std::cout << 'X';
  using namespace std;
  cout << 'X';
  return 0;
}
```

64

We will examine the code moving from the start down to the end of the program, examining fragments of it in turn.

```
#include <iostream>
```

This just includes the iostream library so that we can use `std::cout` to print stuff to the screen.

```
namespace first {
  int first1;
  int x;
}

namespace second {
  int second1;
  int x;
}

namespace first {
  int first2;
}
```

64 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

We create a namespace called *first* and add to it two variables, *first1* and *x*. Then we close it. Then we create a new namespace called *second* and put two variables in it: *second1* and *x*. Then we re-open the `namespace` *first* and add another variable called *first2* to it. A `namespace` can be re-opened in this manner as often as desired to add in extra names.

```
  main(){
1  //first1 = 1;
2  first::first1 = 1;
```

The first line of the main program is commented out because it would cause an error. In order to get at a name from the first `namespace`, we must qualify the variable's name with the name of its `namespace` before it and two colons; hence the second line of the main program is not a syntax error. The name of the variable is in scope: it just has to be referred to in that particular way before it can be used at this point. This therefore cuts up the list of global names into groups, each group with its own prefixing name.

```
3  using namespace first;
4  first1 = 1;
5  x = 1;
6  second::x = 1;
```

The third line of the main program introduces the *using namespace* command. This commands pulls all the names in the first `namespace` into scope. They can then be used in the usual way from there on. Hence the fourth and fifth lines of the program compile without error. In particular, the variable *x* is available now: in order to address the other variable *x* in the second `namespace`, we would call it *second::x* as shown in line six. Thus the two variables called *x* can be separately referred to, as they are on the fifth and sixth lines.

```
7  using namespace second;
8  //x = 1;
9  first::x = 1;
10 second::x = 1;
```

We then pull the declarations in the `namespace` called *second* in, again with the *using namespace* command. The line following is commented out because it is now an error (whereas before it was correct). Since both namespaces have been brought into the global list of names, the variable *x* is now ambiguous, and needs to be talked about only in the qualified manner illustrated in the ninth and tenth lines.

```
11 first2 = 1;
```

The eleventh line of the main program shows that even though *first2* was declared in a separate section of the `namespace` called *first*, it has the same status as the other variables in `namespace` *first*. A `namespace` can be re-opened as many times as you wish. The usual rules of scoping apply, of course: it is not legal to try to declare the same name twice in the same `namespace`.

```
12 //cout << 'X';
13 std::cout << 'X';
14 using namespace std;
15 cout << 'X';
}
```

There is a `namespace` defined in the computer in special group of files. Its name is *std* and all the system-supplied names, such as *cout*, are declared in that `namespace` in a number of different files: it is a very large namespace. Note that the *#include* statement at the very top of the program does not fully bring the `namespace` in: the names are there but must still be referred to in qualified form. Line twelve has to be commented out because currently the system-supplied names like *cout* are not available, except in the qualified form *std::cout* as can be seen in line thirteen. Thus we need a line like the fourteenth line: after that line is written, all the system-supplied names are available, as illustrated in the last line of the program. At this point we have the names of three `namespace` incorporated into the program.

As the example program illustrates, the declarations that are needed are brought in as desired, and the unwanted ones are left out, and can be brought in in a controlled manner using the qualified form with the double colons. This gives the greater control of names needed for large programs. In the example above, we used only the names of variables. However, namespaces also control, equally, the names of procedures and classes, as desired.

3.2 The Compiler

A **COMPILER**[65] is a program that translates a COMPUTER PROGRAM[66] written in one COMPUTER LANGUAGE[67] (the SOURCE CODE[68]) into an equivalent program written in the computer's native MACHINE LANGUAGE[69]. This process of transla-

65 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPILER
66 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPUTER%20PROGRAM
67 HTTP://EN.WIKIBOOKS.ORG/WIKI/PROGRAMMING%20LANGUAGES%20BOOKSHELF
68 Chapter 3.1.2 on page 42
69 HTTP://EN.WIKIPEDIA.ORG/WIKI/MACHINE%20LANGUAGE

tion, that includes several distinct steps is called **compilation**. Since the compiler is a program, itself written in a computer language, the situation may seem a paradox akin to the CHICKEN AND EGG DILEMMA[70]. A compiler may not be created with the resulting compilable language but with a previous available language or even in machine code.

3.2.1 Compilation

The **compilation** output of a compiler is the result from translating or *compiling* a program. The most important part of the output is saved to a file called an OBJECT FILE[71], it consists of the transformation of source files into object files.

> **Note:**
> Some files may be created/needed for a successful compilation, that data is not part of the C++ language or may result from the compilation of external code (an example would be a library), this may depend on the specific compiler you use (MS Visual Studio for example adds several extra files to a project), in that case you should check the documentation or it can part of a specific framework that needs to be accessed. Be aware that some of this constructs may limit the portability of the code.

The instructions of this *compiled* program can then be run (executed) by the computer if the object file is in an executable format. However, there are additional steps that are required for a compilation: preprocessing and linking.

Compile-time

Defines the time and operations performed by a compiler (i.e., *compile-time operations*) during a build (creation) of a program (executable or not). Most of the uses of "static" on the C++ language is directly related to compile-time information.

The operations performed at compile time usually include *lexical analysis*, *syntax analysis*, various kinds of SEMANTIC ANALYSIS[72] (e.g., TYPE CHECKS[73], some

70 HTTP://EN.WIKIPEDIA.ORG/WIKI/CHICKEN%20OR%20THE%20EGG
71 Chapter 3 on page 41
72 HTTP://EN.WIKIPEDIA.ORG/WIKI/SEMANTIC%20ANALYSIS%20%28COMPUTER%20SCIENCE%29
73 HTTP://EN.WIKIPEDIA.ORG/WIKI/DATATYPE

of the TYPE CASTS[74], and INSTANTIATION OF TEMPLATE[75]) and CODE GENERATION[76].

The definition of a programming language will specify compile time requirements that source code must meet to be successfully compiled.

Compile time occurs before LINK TIME[77] (when the output of one or more compiled files are joined together) and runtime (when a program is executed). In some programming languages it may be necessary for some compilation and linking to occur at runtime.

Run-time

Run-time, or **execution time**, starts at the moment the program starts to execute and end as it exits. At this stage the compiler is irrelevant and has no control. This is the most important location in regards to optimizations (a program will only compile once but run many times) and debugging (tracing and interaction will only be possible at this stage). But it is also in run-time that some of the TYPE CASTING MAY OCCUR[78] and that RUN-TIME TYPE INFORMATION (RTTI)[79] has relevance. The concept of runtime will be mentioned again when relevant.

Lexical analysis

This is alternatively known as scanning or *tokenisation*. It happens before syntax analysis and converts the code into TOKENS[80], which are the parts of the code that the program will actually use. The source code as expressed as characters (arranged on lines) into a sequence of special tokens for each reserved keyword, and tokens for data types and identifiers and values. The lexical analyzer is the part of the compiler which removes whitespace and other non compilable characters from the source code. It uses whitespace to separate different tokens, and ignores the whitespace.

To give a simple illustration of the process:

74 Chapter 3.4.14 on page 204
75 HTTP://EN.WIKIPEDIA.ORG/WIKI/INSTANTIATION%20OF%20TEMPLATE
76 HTTP://EN.WIKIPEDIA.ORG/WIKI/CODE%20GENERATION%20%28COMPILER%29
77 HTTP://EN.WIKIPEDIA.ORG/WIKI/LINK%20TIME
78 Chapter 3.4.14 on page 204
79 Chapter 5.5.5 on page 530
80 HTTP://EN.WIKIBOOKS.ORG/WIKI/COMPILER%20CONSTRUCTION%23WHAT%20IS%20A%20TOKEN

```
int main()
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

Depending on the lexical rules used it might be **tokenized** as:

```
1 = string "int"
2 = string "main"
3 = opening parenthesis
4 = closing parenthesis
5 = opening brace
6 = string "std"
7 = namespace operator
8 = string "cout"
9 = << operator
10 = string ""hello world""
11 = string "endl"
12 = semicolon
13 = string "return"
14 = number 0
15 = closing brace
```

And so for this program the lexical analyzer might send something like:

```
1 2 3 4 5 6 7 8 9 10 9 6 7 11 12 13 14 12 15
```

To the syntactical analyzer, which is talked about next, to be parsed. It is easier for the syntactical analyzer to apply the rules of the language when it can work with numerical values and can distinguish between language syntax (such as the semicolon) and everything else, and knows what data type each thing has.

Syntax analysis

This step (also called sometimes syntax checking) ensures that the code is valid and will sequence into an executable program. The syntactical analyzer applies rules to the code, checking to make sure that each opening brace has a corresponding closing brace, and that each declaration has a type, and that the type exists, and that.... syntax analysis is more complicated than lexical analysis =).

As an example:

```
int main()
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

- The syntax analyzer would first look at the string "int", check it against defined keywords, and find that it is a type for integers. *The analyzer would then look at the next token as an identifier, and check to make sure that it has used a valid identifier name.
- It would then look at the next token. Because it is an opening parenthesis it will treat "main" as a function, instead of a declaration of a variable if it found a semicolon or the initialization of an integer variable if it found an equals sign.
- After the opening parenthesis it would find a closing parenthesis, meaning that the function has 0 parameters.
- Then it would look at the next token and see it was an opening brace, so it would think that this was the implementation of the function main, instead of a declaration of main if the next token had been a semicolon, even though you can not declare main in c++. It would probably create a counter also to keep track of the level of the statement blocks to make sure the braces were in pairs. *After that it would look at the next token, and probably not do anything with it, but then it would see the :: operator, and check that "std" was a valid `namespace`.
- Then it would see the next token "cout" as the name of an identifier in the `namespace` "std", and see that it was a template.
- The analyzer would see the << operator next, and so would check that the << operator could be used with cout, and also that the next token could be used with the << operator.
- The same thing would happen with the next token after the ""hello world"" token. Then it would get to the "std" token again, look past it to see the :: operator token and check that the `namespace` existed again, then check to see if "endl" was in the `namespace`.
- Then it would see the semicolon and so it would see that as the end of the statement.
- Next it would see the keyword `return`, and then expect an integer value as the next token because main returns an integer, and it would find 0, which is an integer.
- Then the next symbol is a semicolon so that is the end of the statement.
- The next token is a closing brace so that is the end of the function. And there are no more tokens, so if the syntax analyzer did not find any errors with the code,

it would send the tokens to the compiler so that the program could be converted to machine language.

This is a simple view of syntax analysis, and real syntax analyzers do not really work this way, but the idea is the same.

Here are some keywords which the syntax analyzer will look for to make sure you are not using any of these as identifier names, or to know what type you are defining your variables as or what function you are using which is included in the C++ language.

Compile speed

There are several factors that dictate how fast a compilation proceeds, like:

- Hardware
  - Resources (Slow CPU, low memory and even a slow HDD can have an influence)
- Software
  - The compiler itself, new is always better, but may depend on how portable you want the project to be.
  - The design selected for the program (structure of object dependencies, includes) will also factor in.

Experience tells that most likely if you are suffering from slow compile times, the program you are trying to compile is poorly designed, take the time to structure your own code to minimize re-compilation after changes. Large projects will always compile slower. Use pre-compiled headers and external header guards. We will discuss ways to reduce compile time in the OPTIMIZATION[81] Section of this book.

3.2.2 Where to get a compiler

When you select your compiler you must take in consideration your system OS, your personal preferences and the documentation that you can get on using it.

Most compilers today are free and many open source platforms already include one (mostly GCC), there are also various IDEs available.

81 Chapter 6.8.3 on page 635

In case you don't have, want or need a compiler installed on you machine, you can use a WEB free compiler available at HTTP://IDEONE.COM[82] (or HTTP://CODEPAD.ORG[83] but you will have to change the code not to require interactive input). You can always get one locally if you need it.

IDE (Integrated development environment)

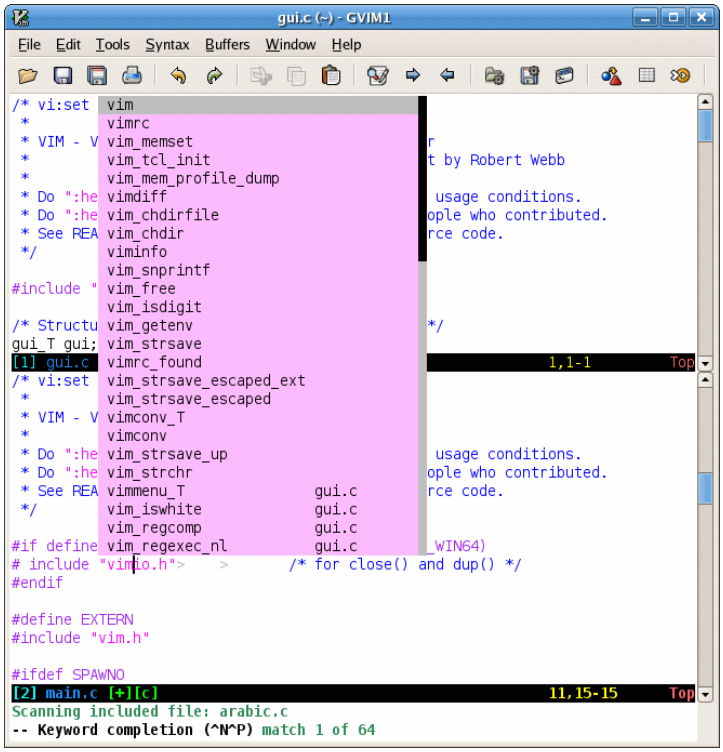


Figure 7: Graphical Vim under GTK2[a]

a HTTP://EN.WIKIPEDIA.ORG/WIKI/GTK%2B

INTEGRATED DEVELOPMENT ENVIRONMENT[84] is a software development system, that often includes an editor, compiler and debugger in an integrated package that is distributed together. Some IDEs will require the user to make the integra-

82 HTTP://IDEONE.COM

83 HTTP://CODEPAD.ORG

84 HTTP://EN.WIKIPEDIA.ORG/WIKI/INTEGRATED%20DEVELOPMENT%20ENVIRONMENT

tion of the components themselves, and others will refer as the IDE to the set of separated tools they use for programming.

A good IDE is one that permits the programmer to use it to abstract and accelerate some of the more common tasks and at the same time provide some help in reading and managing the code. Except for the compiler the C++ Standard has no control over the different implementations. Most IDEs are visually oriented, especially the new ones, they will offer graphical debuggers and other visual aids, but some people will still prefer the visual simplicity offered by potent text editors like VIM[85] or EMACS[86].

When selecting an IDE, remember that you are also investing time to become proficient in its use, completeness, stability and portability across OSs will be important.

For Microsoft Windows, you have also the Microsoft Visual Studio Express, currently freely available (but with reduced functionalities), it includes a C++ compiler that can be used from the command line or the supplied IDE.

In the book APPENDIX B:EXTERNAL REFERENCES[87] you will find references to other freely available compilers and IDEs you can use.

GCC[88]

One of most mature and compatible compilers is GCC. Also known as The GNU Compiler Collection is a free set of compilers developed by the Free Software Foundation, with RICHARD STALLMAN[89] as one of the main architects.

There are many different pre-compiled GCC binaries on the Internet; some popular choices are listed below (with detailed steps for installation).You can easily find information on the GCC website on how to do it under another OS.

85 HTTP://EN.WIKIBOOKS.ORG/WIKI/LEARNING%20THE%20VI%20EDITOR%2FVIM

86 HTTP://EN.WIKIPEDIA.ORG/WIKI/EMACS

87 Chapter 8.2 on page 646

88 HTTP://EN.WIKIPEDIA.ORG/WIKI/GNU%20COMPILER%20COLLECTION

89 HTTP://EN.WIKIPEDIA.ORG/WIKI/RICHARD%20STALLMAN

Note:
Is often common that the implementation language of a compiler to be C (since it is normally first the system language above assembly that new systems implement). GCC has, since the end of May 2005, GOT THE GREEN LIGHT[a] to start moving the core code-base to C++. Considering that this is the most common used compiler and an open source implementation, it was an extremely positive step to the compiler and the language in general.

a HTTP://ARTICLE.GMANE.ORG/GMANE.COMP.GCC.DEVEL/114407

On Windows
Cygwin:

1. Go to HTTP://WWW.CYGWIN.COM[90] and click on the "Install Cygwin Now" button in the upper right corner of the page.
2. Click "run" in the window that pops up, and click "next" several times, accepting all the default settings.
3. Choose any of the Download sites ("ftp.easynet.be", etc.) when that window comes up; press "next" and the Cygwin installer should start downloading.
4. When the "Select Packages" window appears, scroll down to the heading "Devel" and click on the "+" by it. In the list of packages that now displays, scroll down and find the "gcc-c++" package; this is the compiler. Click once on the word "Skip", and it should change to some number like "3.4" etc. (the version number), and an "X" will appear next to "gcc-core" and several other required packages that will now be downloaded.
5. Click "next" and the compiler as well as the Cygwin tools should start downloading; this could take a while. While you're waiting, go to HTTP://WWW.CRIMSONEDITOR.COM[91] and download that free programmer's editor; it's powerful yet easy to use for beginners.
6. Once the Cygwin downloads are finished and you have clicked "next", etc. to finish the installation, double-click the Cygwin icon on your desktop to begin the Cygwin "command prompt". Your home directory will automatically be set up in the Cygwin folder, which now should be at "C:\cygwin" (the Cygwin folder is in some ways like a small Unix/Linux computer on your Windows machine -- not technically of course, but it may be helpful to think of it that way).

90 HTTP://WWW.CYGWIN.COM
91 HTTP://WWW.CRIMSONEDITOR.COM

7. Type "g++" at the Cygwin prompt and press "enter"; if "g++: no input files" or something like it appears you have succeeded and now have the gcc C++ compiler on your computer (and congratulations -- you have also just received your first error message!).

MinGW + DevCpp-IDE

1. Go to HTTP://WWW.BLOODSHED.NET/DEVCPP.HTML,[92] choose the version you want (eventually scrolling down), and click on the appropriate download link! For the most current version, you will be redirected to http://www.bloodshed.net/dev/devcpp.html
2. Scroll down to read the license and then to the download links. Download a version *with Mingw/GCC*. It's much easier than to do this assembling yourself. With a very short delay (only some days) you will always get the most current version of MinGW packaged with the devcpp IDE. It's absolutely the same as with manual download of the required modules.
3. You get an executable that can be executed at user level under any WinNT version. If you want it to be setup for all users, however, you need admin rights. It will install devcpp and mingw in folders of your wish.
4. Start the IDE and experience your first project!
   You will find something mostly similar to MSVC, including menu and button placement. Of course, many things are somewhat different if you were familiar with the former, but it's as simple as a handful of clicks to let your first program run.

For DOS
DJGPP:

- Go to DELORIE SOFTWARE[93] and download the GNU C++ compiler and other necessary tools. The site provides a *Zip Picker*[94] in order to help identify which files you need, which is available from the main page.
- Use unzip32 or other extraction utility to place files into the directory of your choice (i.e. C:\DJGPP).
- Set the envionment variables to configure DJGPP for compilation, by either adding lines to autoexec.bat or a custom batch file:

```
set PATH=C:\DJGPP\BIN;%PATH%
set DJGPP=C:\DJGPP\DJGPP.ENV
```

92 HTTP://WWW.BLOODSHED.NET/DEVCPP.HTML,
93 HTTP://WWW.DELORIE.COM
94 HTTP://WWW.DELORIE.COM/DJGPP/ZIP-PICKER.HTML

- If you are running MS-DOS or Windows 3.1, you need to add a few lines to config.sys if they are not already present:

```
shell=c:\dos\command.com c:\dos /e:2048 /p
files=40
fcbs=40,0
```

Note: The GNU C++ compiler under DJGPP is named *gpp*.

For Linux

- For GENTOO[95], GCC C++ is part of the system core (since everything in Gentoo is compiled)
- For REDHAT[96], get a gcc-c++ RPM[97], e.g. using Rpmfind and then install (as root) using `rpm -ivh gcc-c++-`*version-release*`.`*arch*`.rpm`
- For FEDORA CORE[98], install the GCC C++ compiler (as root) by using YUM[99] `install gcc-c++`
- For MANDRAKE[100], install the GCC C++ compiler (as root) by using URPMI[101] `gcc-c++`
- For DEBIAN[102], install the GCC C++ compiler (as root) by using APT-GET[103] `install g++`
- For UBUNTU[104], install the GCC C++ compiler by using `sudo apt-get install g++`
- For OPENSUSE[105], install the GCC C++ compiler (as root) by using ZYPPER[106] `in gcc-c++`
- If you cannot become root, get the tarball from ftp://ftp.gnu.org/ and follow the instructions in it to compile and install in your home directory.

95 HTTP://EN.WIKIPEDIA.ORG/WIKI/GENTOO%20LINUX
96 HTTP://EN.WIKIPEDIA.ORG/WIKI/REDHAT
97 HTTP://EN.WIKIPEDIA.ORG/WIKI/RPM%20PACKAGE%20MANAGER
98 HTTP://EN.WIKIPEDIA.ORG/WIKI/FEDORA%20CORE
99 HTTP://EN.WIKIPEDIA.ORG/WIKI/YELLOW%20DOG%20UPDATER%20MODIFIED
100 HTTP://EN.WIKIPEDIA.ORG/WIKI/MANDRAKE
101 HTTP://EN.WIKIPEDIA.ORG/WIKI/URPMI
102 HTTP://EN.WIKIPEDIA.ORG/WIKI/DEBIAN
103 HTTP://EN.WIKIPEDIA.ORG/WIKI/APT
104 HTTP://EN.WIKIPEDIA.ORG/WIKI/UBUNTU
105 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPENSUSE
106 HTTP://EN.WIKIPEDIA.ORG/WIKI/ZYPP

For Mac OS X
XCODE[107] has GCC C++ compiler bundled. It can be invoked from the Terminal in the same way as Linux, but can also be compiled in one of XCode's projects.

3.2.3 The Preprocessor

The PREPROCESSOR[108] is either a separate program invoked by the COMPILER[109] or part of the compiler itself. It performs intermediate operations that modify the original source code and internal compiler options before the compiler tries to compile the resulting source code.

The instructions that the preprocessor PARSES[110] are called **directives** and come in two forms: preprocessor and compiler directives. **Preprocessor directives** direct the preprocessor on how it should process the source code, and **compiler directives** direct the compiler on how it should modify internal compiler options. Directives are used to make writing source code easier (by making it more portable, for instance) and to make the source code more understandable. They are also the only valid way to make use of facilities (classes, functions, templates, etc.) provided by the C++ Standard Library.

Note:
Check the documentation of your compiler/preprocessor for information on how it implements the preprocessing phase and for any additional features not covered by the standard that may be available. For in depth information on the subject of parsing, you can read "COMPILER CONSTRUCTION[a]" (http://en.wikibooks.org/wiki/Compiler_Construction)

a HTTP://EN.WIKIBOOKS.ORG/WIKI/COMPILER%20CONSTRUCTION

All directives start with '#' at the beginning of a line. The standard directives are:

- #define
- #elif
- #else
- #endif
- #error
- #if
- #ifdef
- #ifndef
- #include
- #line
- #pragma
- #undef

107 HTTP://EN.WIKIPEDIA.ORG/WIKI/XCODE
108 HTTP://EN.WIKIPEDIA.ORG/WIKI/PREPROCESSOR
109 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPILER
110 HTTP://EN.WIKIPEDIA.ORG/WIKI/PARSING

Inclusion of Header Files (#include)

The **#include** directive allows a programmer to include contents of one file inside another file. This is commonly used to separate information needed by more than one part of a program into its own file so that it can be included again and again without having to re-type all the source code into each file.

C++ generally requires you to *declare* what will be used before using it. So, files called **HEADERS**[111] usually include declarations of what will be used in order for the compiler to successfully compile source code. This is further explained in the FILE ORGANIZATION SECTION[112] of the book. The **standard library** (the repository of code that is available with every standards-compliant C++ compiler) and 3rd party libraries make use of headers in order to allow the inclusion of the needed declarations in your source code, allowing you to make use of features or resources that are not part of the language itself.

The first lines in any source file should usually look something like this:

```
#include <iostream>
#include "other.h"
```

The above lines cause the contents of the files *iostream* and *other.h* to be included for use in your program. Usually this is implemented by just inserting into your program the contents of *iostream* and *other.h*. When angle brackets (<>) are used in the directive, the preprocessor is instructed to search for the specified file in a compiler-dependent location. When double quotation marks (`" "`) are used, the preprocessor is expected to search in some additional, usually user-defined, locations for the header file and to fall back to the standard include paths only if it is not found in those additional locations. Commonly when this form is used, the preprocessor will also search in the same directory as the file containing the `#include` directive.

The `iostream` header contains various declarations for input/output (I/O) using an abstraction of I/O mechanisms called **streams**. For example there is an output stream object called `std::cout` (where "cout" is short for "console output") which is used to output text to the standard output, which usually displays the text on the computer screen.

111 HTTP://EN.WIKIPEDIA.ORG/WIKI/HEADER%20%28INFORMATION%20TECHNOLOGY%29

112 Chapter 3.1.5 on page 49

Note:
When including standard libraries, compilers are allowed to make an exception as to whether a header file by a given name actually exists as a physical file or is simply a logical entity that causes the preprocessor to modify the source code, with the same end result as if the entity existed as a physical file. Check the documentation of your preprocessor/compiler for any vendor-specific implementation of the #include directive and for specific search locations of standard and user-defined headers. This can lead to portability problems and confusion.

A list of standard C++ header files is listed below:

Standard Template Library

113 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23algorithm
114 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23bitset
115 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23complex
116 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23deque
117 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23exception
118 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23fstream
119 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23functional
120 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23iomanip
121 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23ios
122 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23iosfwd
123 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23iostream
124 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23istream
125 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23iterator
126 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23limits
127 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23list
128 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23locale
129 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23map
130 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23memory
131 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23new
132 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23numeric
133 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23ostream
134 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23queue
135 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23set
136 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23sstream
137 http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2Fheaders%23stack

- ALGORITHM[113]
- BITSET[114]
- COMPLEX[115]
- DEQUE[116]
- EXCEPTION[117]
- FSTREAM[118]
- FUNCTIONAL[119]
- IOMANIP[120]
- IOS[121]
- IOSFWD[122]
- IOSTREAM[123]
- ISTREAM[124]
- ITERATOR[125]
- LIMITS[126]
- LIST[127]
- LOCALE[128]
- MAP[129]
- MEMORY[130]
- NEW[131]
- NUMERIC[132]
- OSTREAM[133]
- QUEUE[134]
- SET[135]
- SSTREAM[136]
- STACK[137]
- STDEXCEPT[138]
- STREAMBUF[139]
- STRING[140]
- STRSTREAM[141]
- TYPEINFO[142]
- UTILITY[143]
- VALARRAY[144]
- VECTOR[145]

and the

Standard C Library

138 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23STDEXCEPT

139 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23STREAMBUF

140 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23STRING

141 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23STRSTREAM

142 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23TYPEINFO

143 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23UTILITY

144 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23VALARRAY

145 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23VECTOR

- CASSERT[146]
- CCTYPE[147]
- CERRNO[148]
- CFLOAT[149]
- CISO646[150]
- CLIMITS[151]
- CLOCALE[152]
- CMATH[153]
- CSETJMP[154]
- CSIGNAL[155]
- CSTDARG[156]
- CSTDDEF[157]
- CSTDIO[158]
- CSTDLIB[159]
- CSTRING[160]
- CTIME[161]
- CWCHAR[162]
- CWCTYPE[163]

Everything inside C++'s standard library is kept in the `std::` namespace.

146 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CASSERT
147 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CCTYPE
148 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CERRNO
149 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CFLOAT
150 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CISO646
151 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CLIMITS
152 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CLOCALE
153 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CMATH
154 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CSETJMP
155 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CSIGNAL
156 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CSTDARG
157 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CSTDDEF
158 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CSTDIO
159 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CSTDLIB
160 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CSTRING
161 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CTIME
162 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CWCHAR
163 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23CWCTYPE

Old compilers may include headers with a `.h` suffix (e.g. the non-standard `<iostream.h>` vs. the standard `<iostream>`) instead of the standard headers. These names were common before the standardization of C++ and some compilers still include these headers for backwards compatibility. Rather than using the `std::` namespace, these older headers pollute the global namespace and may otherwise only implement the standard in a limited way.

Some vendors use the SGI[164] STL[165] headers. This was the first implementation of the standard template library.

Non-standard but somewhat common C++ libraries

- `STDIOSTREAM.H`[166,167]
- `STREAM.H`[168,169]
- `STRSTREAM.H`[170,171]

Note:
Before standardization of the headers, they were presented as separated files, like <iostream.h> and so on. This is probably still a requirement on very old (non-standards-compliant) compilers, but newer compilers will accept both methods. There is also no requirement in the standard that headers should exist in a file form. The old method of referring to standard libraries as separate files is obsolete.

#pragma

The **pragma** (pragmatic information) directive is part of the standard, but the meaning of any pragma directive depends on the software implementation of the standard that is used.

Pragma directives are used within the source program.

164 HTTP://EN.WIKIPEDIA.ORG/WIKI/SILICON%20GRAPHICS
165 Chapter 5.1.5 on page 499
166 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23STDIOSTREAM.H
167 Streams based on FILE* from stdio.h.
168 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23STREAM.H
169 Precursor to iostream. Old stream library mostly included for backwards compatibility even with old compilers.
170 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FHEADERS%23STRSTREAM.H
171 Uses **char*** whereas sstream uses string. Prefer the standard library sstream.

```
#pragma token(s)
```

You should check the software implementation of the C++ standard you intend to use for a list of the supported tokens.

For example, one of the most widely used preprocessor pragma directives, `#pragma once`, when placed at the beginning of a header file, indicates that the file where it resides will be skipped if included several times by the preprocessor.

Note:
Another method exists, commonly referred to as **include guards**, that provides this same functionality but uses other include directives.

In the GCC documentation, `#pragma once` has been described as an obsolete preprocessor directive.

Macros

The C++ preprocessor includes facilities for defining "macros", which roughly means the ability to replace a use of a named macro with one or more tokens. This has various uses from defining simple constants (though **const** is more often used for this in C++), conditional compilation, code generation and more -- macros are a powerful facility, but if used carelessly can also lead to code that is hard to read and harder to debug!

Note:
Macros do not depend only on the C++ Standard or your actions. They may exist due to the use of external frameworks, libraries or even due the compiler you are using and the specific OS. We will not cover that information on this book but you may find more information in the *Pre-defined C/C++ Compiler Macros* page at (HTTP://PREDEF.SOURCEFORGE.NET/[a]) the project maintains a complete list of macros that are compiler and OS agnostic.

a HTTP://PREDEF.SOURCEFORGE.NET/

#define and #undef

The **#define** directive is used to define values or macros that are used by the preprocessor to manipulate the program source code before it is compiled:

```
#define USER_MAX (1000)
```

The **#undef** directive deletes a current macro definition:

```
#undef USER_MAX
```

It is an error to use **#define** to change the definition of a macro, but it is not an error to use **#undef** to try to undefine a macro name that is not currently defined. Therefore, if you need to override a previous macro definition, first **#undef** it, and then use **#define** to set the new definition.

Note:
Because preprocessor definitions are substituted before the compiler acts on the source code, any errors that are introduced by `#define` are difficult to trace. For example using value or macro names that are the same as some existing identifier can create subtle errors, since the preprocessor will substitute the identifier names in the source code.
Today, for this reason, `#define` is primarily used to handle compiler and platform differences. E.g, a define might hold a constant which is the appropriate error code for a system call. The use of `#define` should thus be limited unless absolutely necessary; typedef statements, constant variables, enums, templates and INLINE FUNCTIONS[a] can often accomplish the same goal more efficiently and safely.
By convention, values defined using `#define` are named in uppercase with "_" separators, this makes it clear to readers that the values is not alterable and in the case of macros, that the construct requires care. Although doing so is not a requirement, it is considered very bad practice to do otherwise. This allows the values to be easily identified when reading the source code.
Try to use `const` and `inline` instead of `#define`.

a Chapter 3.7 on page 229

\ (line continuation)
If for some reason it is needed to break a given statement into more than one line, use the **** (backslash) symbol to "escape" the line ends. For example,

```
#define MULTIPLELINEMACRO \
        will use what you write here \
        and here etc...
```

is equivalent to

```
#define MULTIPLELINEMACRO will use what you write here and here etc...
```

because the preprocessor joins lines ending in a backslash ("\") to the line after them. That happens even before directives (such as #define) are processed, so it works for just about all purposes, not just for macro definitions. The backslash is sometimes said to act as an "escape" character for the newline, changing its interpretation.

In some (fairly rare) cases macros can be more readable when split across multiple lines. Good modern C++ code will use macros only sparingly, so the need for multi-line macro definitions will not arise often.

It is certainly possible to overuse this feature. It is quite legal but entirely indefensible, for example, to write

```
int ma\
in//ma/
()/*ma/
in/*/{}
```

That is an abuse of the feature though: while an escaped newline *can* appear in the middle of a token, there should never be any reason to use it there. Do not try to write code that looks like it belongs in the International Obfuscated C Code Competition.

Warning: there is one occasional "gotcha" with using escaped newlines: if there are any invisible characters after the backslash, the lines will not be joined, and there will almost certainly be an error message produced later on, though it might not be at all obvious what caused it.

Function-like Macros

Another feature of the #define command is that it can take arguments, making it rather useful as a pseudo-function creator. Consider the following code:

```
#define ABSOLUTE_VALUE( x ) ( ((x) < 0) ? -(x) : (x) )
// ...
int x = -1;
while( ABSOLUTE_VALUE( x ) ) {
// ...
}
```

Note:
It is generally a good idea to use extra parentheses for macro parameters, it avoids the parameters from being parsed in a unintended ways. But there are some exceptions to consider:

1. Since comma operator have lower precedence than any other, this removes the possibility of problems, no need for the extra parentheses.
2. When concatenating tokens with the ## operator, converting to strings using the # operator, or concatenating adjacent string literals, parameters cannot be individually parenthesized.

Notice that in the above example, the variable "x" is always within its own set of parentheses. This way, it will be evaluated in whole, before being compared to 0 or multiplied by -1. Also, the entire macro is surrounded by parentheses, to prevent it from being contaminated by other code. If you're not careful, you run the risk of having the compiler misinterpret your code.

Macros replace each occurrence of the macro parameter used in the text with the literal contents of the macro parameter without any validation checking. Badly written macros can result in code which will not compile or creates hard to discover bugs. Because of side-effects it is considered a very bad idea to use macro functions as described above. However as with any rule, there may be cases where macros are the most efficient means to accomplish a particular goal.

```
int z = -10;
int y = ABSOLUTE_VALUE( z++ );
```

If ABSOLUTE_VALUE() was a real function 'z' would now have the value of '-9', but because it was an argument in a macro **z++** was expanded 3 times (in this case) and thus (in this situation) executed twice, setting z to -8, and y to 9. In similar cases it is very easy to write code which has "undefined behavior", meaning that what it does is completely unpredictable in the eyes of the C++ Standard.

```
// ABSOLUTE_VALUE( z++ ); expanded
( ((z++) < 0 ) ? -(z++) : (z++) );
```

Note:
With the GCC compiler extension called "statement expression" (not standard C++), it is allowed to use statements in an expression, please consult the compiler manual for other considerations, it becomes then possible to only evaluate it once:

```
define ABSOLUTE_VALUE( x ) ( { typeof (x) temp = (x); (temp < 0) ? -temp : temp; } )
```

Using inlined templated functions may then be an alternative to macros, removing the problem of side effects inside the argument to the macro.
It is generally good idea to stay away from compiler specific extensions, unless the dependency is planed for.

and

```
// An example on how to use a macro correctly

#include <iostream>

#define SLICES 8
#define PART(x) ( (x) / SLICES ) // Note the extra parentheses around '''x'''

int main() {
   int b = 10, c = 6;

   int a = PART(b + c);
   std::cout << a;

   return 0;
}
```

-- the result of "a" should be "2" (b + c passed to PART -> ((b + c) / SLICES) -> result is "2")

Note:
Variadic Macros

A variadic macro is a feature of the preprocessor whereby a macro is declared to accept a varying number of arguments (similar to a variadic function).
They are currently not part of the C++ programming language, though many recent C++ implementations support variable-argument macros as an extension (ie: GCC, MS Visual Studio C++), and it is expected that variadic macros may be added to C++ at a later date.
Variable-argument macros were introduced in the ISO/IEC 9899:1999 (C99) revision of the C Programming Language standard in 1999.

and

The **#** and **##** operators are used with the `#define` macro. Using # causes the first argument after the **#** to be returned as a string in quotes. For example

```
#define as_string( s ) # s
```

will make the compiler turn

```
std::cout << as_string( Hello  World! ) << std::endl;
```

into

```
std::cout << "Hello World!" << std::endl;
```

Note:
Observe the leading and trailing whitespace from the argument to `#` is removed, and consecutive sequences of whitespace between tokens are converted to single spaces.

Using **##** concatenates what's before the **##** with what's after it; the result must be a well-formed preprocessing token. For example

```
#define concatenate( x, y ) x ## y
...
int xy = 10;
...
```

will make the compiler turn

```
std::cout << concatenate( x, y ) << std::endl;
```

into

```
std::cout << xy << std::endl;
```

which will, of course, display `10` to standard output.

String literals cannot be concatenated using **##**, but the good news is that this is not a problem: just writing two adjacent string literals is enough to make the preprocessor concatenate them.

The dangers of macros

To illustrate the dangers of macros, consider this naive macro

```
#define MAX(a,b) a>b?a:b
```

and the code

```
i = MAX(2,3)+5;
j = MAX(3,2)+5;
```

Take a look at this and consider what the value after execution might be. The statements are turned into

```
int i = 2>3?2:3+5;
int j = 3>2?3:2+5;
```

Thus, after execution `i=8` and `j=3` instead of the expected result of `i=j=8`! This is why you were cautioned to use an extra set of parenthesis above, but even with these, the road is fraught with dangers. The alert reader might quickly realize that if `a,b` contains expressions, the definition must parenthesize every use of `a,b` in the macro definition, like this:

```
#define MAX(a,b) ((a)>(b)?(a):(b))
```

This works, provided `a,b` have no side effects. Indeed,

```
i = 2;
j = 3;
k = MAX(i++, j++);
```

would result in `k=4`, `i=3` and `j=5`. This would be highly surprising to anyone expecting `MAX()` to behave like a function.

So what is the correct solution? The solution is not to use macro at all. A global, inline function, like this `inline max(int a, int b) { return a>b?a:b }` has none of the pitfalls above, but will not work with all types. A template (see below) takes care of this `template<typename T> inline max(const T& a, const T& b) { return a>b?a:b }` Indeed, this is (a variation of) the definition used in STL library for std::max(). This library is included with all conforming C++ compilers, so the ideal solution would be to use this.

```
std::max(3,4);
```

Another danger on working with macro is that they are excluded form type checking. In the case of the MAX macro, if used with a string type variable, it will not generate a compilation error.

```
MAX("hello","world")
```

It is then preferable to use a inline function, which will be type checked. Permitting the compiler to generate a meaningful error message if the inline function is used as stated above.

String literal concatenation

One minor function of the preprocessor is in joining strings together, "string literal concatenation" -- turning code like

```
std::cout << "Hello " "World!\n";
```

into

```
std::cout << "Hello World!\n";
```

Apart from obscure uses, this is most often useful when writing long messages, as it is not legal in C++ (at this time) to have a string literal which spans multiple lines in your source code (i.e., one which has a newline character inside it). It also

helps to keep program lines down to a reasonable length; we can write

```
function_name("This is a very long string literal, which would not fit "
              "onto a single line very nicely -- but with string literal "
              "concatenation, we can split it across multiple lines and "
              "the preprocessor will glue the pieces together");
```

Note that this joining happens before compilation; the compiler sees only one string literal here, and there's no work done at runtime, i.e., your program will not run any slower at all because of this joining together of strings.

Concatenation also applies to wide string literals (which are prefixed by an L):

```
L"this " L"and " L"that"
```

is converted by the preprocessor into

```
L"this and that".
```

Note:
For completeness, note that C99 has different rules for this than C++98, and that C++0x seems almost certain to match C99's more tolerant rules, which allow joining of a narrow string literal to a wide string literal, something which was not valid in C++98.

Conditional compilation

Conditional compilation is useful for two main purposes:

- To allow certain functionality to be enabled/disabled when compiling a program
- To allow functionality to be implemented in different ways, such as when compiling on different platforms

It is also used sometimes to temporarily "comment-out" code, though using a version control system is often a more effective way to do so.

- **Syntax**:

```
#if condition
  statement(s)
#elif condition2
```

```
  statement(s)
...
#elif condition
  statement(s)
#else
  statement(s)
#endif

#ifdef defined-value
  statement(s)
#else
  statement(s)
#endif

#ifndef defined-value
  statement(s)
#else
  statement(s)
#endif
```

#if

The **#if** directive allows compile-time conditional checking of preprocessor values such as created with #DEFINE[172]. If *condition* is non-zero the preprocessor will include all *statement(s)* up to the **#else**, **#elif** or **#endif** directive in the output for processing. Otherwise if the **#if** *condition* was false, any **#elif** directives will be checked in order and the first *condition* which is true will have its *statement(s)* included in the output. Finally if the *condition* of the **#if** directive and any present **#elif** directives are all false the *statement(s)* of the **#else** directive will be included in the output if present; otherwise, nothing gets included.

The expression used after **#if** can include boolean and integral constants and arithmetic operations as well as macro names. The allowable expressions are a subset of the full range of C++ expressions (with one exception), but are sufficient for many purposes. The one extra operator available to **#if** is the **defined** operator, which can be used to test whether a macro of a given name is currently defined.

#ifdef and #ifndef

The **#ifdef** and **#ifndef** directives are short forms of **'#if** defined(*defined-value*)' and **'#if** !defined(*defined-value*)' respectively. **defined**(*identifier*) is valid in any expression evaluated by the preprocessor, and returns true (in this context, equivalent to 1) if a preprocessor variable by the name *identifier* was defined with #define

172 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23%23DEFINE%20AND%20%23UNDEF

and false (in this context, equivalent to 0) otherwise. In fact, the parentheses are optional, and it is also valid to write **defined** *identifier* without them.

(Possibly the most common use of **#ifndef** is in creating "include guards" for header files, to ensure that the header files can safely be included multiple times. This is explained in the section on header files.)

#endif

The **#endif** directive ends **#if**, **#ifdef**, **#ifndef**, **#elif** and **#else** directives.

- **Example**:

```
#if defined(__BSD__) || defined(__LINUX__)
#include <unistd.h>
#endif
```

This can be used for example to provide multiple platform support or to have one common source file set for different program versions. Another example of use is using this instead of the (non-standard) **#pragma once**.

- **Example**:

foo.hpp:

```
#'''ifndef''' FOO_HPP
# '''define''' FOO_HPP

 // code here...

#'''endif''' // FOO_HPP
```

bar.hpp:

```
#'''include''' "foo.h"

 // code here...
```

foo.cpp:

```
#'''include''' "foo.hpp"
#'''include''' "bar.hpp"

 // code here
```

When we compile **foo.cpp**, only one copy of **foo.hpp** will be included due to the use of include guard. When the preprocessor reads the line #**include** `"foo.hpp"`, the content of **foo.hpp** will be expanded. Since this is the first time which **foo.hpp** is read (and assuming that there is no existing declaration of macro **FOO_HPP**) **FOO_HPP** will not yet be declared, and so the code will be included normally.

When the preprocessor read the line `#include "bar.hpp"` in foo.cpp, the content of **bar.hpp** will be expanded as usual, and the file **foo.h** will be expanded again. Owing to the previous declaration of **FOO_HPP**, no code in **foo.hpp** will be inserted. Therefore, this can achieve our goal - avoiding the content of the file being included more than one time.

Compile-time warnings and errors

- **Syntax**:

```
#warning message
#error message
```

#error and #warning

The **#error** directive causes the compiler to stop and spit out the line number and a message given when it is encountered. The **#warning** directive causes the compiler to spit out a warning with the line number and a message given when it is encountered. These directives are mostly used for debugging.

> **Note:**
> **#error** is part of Standard C++, whereas **#warning** is not (though it is widely supported).

- **Example**:

```
#if defined(__BSD___)
#warning Support for BSD is new and may not be stable yet
#endif

#if defined(__WIN95__)
#error Windows 95 is not supported
#endif
```

Source file names and line numbering macros

The current filename and line number where the preprocessing is being performed can be retrieved using the predefined macros __FILE__and __LINE__. Line numbers are measured *before* any escaped newlines are removed. The current values of __FILE__and __LINE__can be overridden using the **#line** directive; it is very

rarely appropriate to do this in hand-written code, but can be useful for code generators which create C++ code base on other input files, so that (for example) error messages will refer back to the original input files rather than to the generated C++ code.

3.2.4 Linker

The **linker** is a program that makes executable files. The linker resolves linkage issues, such as the use of symbols or identifiers which are defined in one translation unit and are needed from other translation units. Symbols or identifiers which are needed outside a single translation unit have *external linkage*. In short, the linker's job is to resolve references to undefined symbols by finding out which other object defines a symbol in question, and replacing placeholders with the symbol's address. Of course, the process is more complicated than this; but the basic ideas apply.

Linkers can take objects from a collection called a library. Depending on the library (system or language or external libraries) and options passed, they may only include its symbols that are referenced from other object files or libraries. Libraries for diverse purposes exist, and one or more system libraries are usually linked in by default. We will take a closer look into libraries on the LIBRARIES SECTION[173] of this book.

Linking

The process of connecting or combining object files produced by a compiler with the libraries necessary to make a working executable program (or a library) is called *linking*. *Linkage* refers to the way in which a program is built out of a number of TRANSLATION UNITS[174].

C++ programs can be compiled and linked with programs written in other languages, such as C, Fortran, assembly language, and Pascal.

- The appropriate compiler compiles each module separately. A C++ compiler compiles each ".cpp" file into a ".o" file, an assembler assembles each ".asm" file into a ".o" file, a Pascal compiler compiles each ".pas" file into a ".o" file, etc.

173 Chapter 6.3.3 on page 584

174 HTTP://EN.WIKIPEDIA.ORG/WIKI/TRANSLATION%20UNIT%20%28PROGRAMMING%29

- The linker links all the ".o" files together in a separate step, creating the final executable file.

Linkage

Every function has either external or internal linkage.

A function with internal linkage is only visible inside one translation unit. When the compiler compiles a function with internal linkage, the compiler writes the machine code for that function at some address and puts that address in all calls to that function (which are all in that one translation unit), but strips out all mention of that function in the ".o" file. If there is some call to a function that apparently has internal linkage, but doesn't appear to be defined in this translation unit, the compiler can immediately tell the programmer about the problem (error). If there is some function with internal linkage that never gets called, the compiler can do "dead code elimination" and leave it out of the ".o" file.

The linker never hears about those functions with internal linkage, so it knows nothing about them.

A function declared with external linkage is visible inside several translation units. When a compiler compiles a call to that function in one translation unit, it does not have any idea where that function is, so it leaves a placeholder in all calls to that function, and instructions in the ".o" file to replace that placeholder with the address of a function with that name. If that function is never defined, the compiler can't possibly know that, so the programmer doesn't get a warning about the problem (error) until much later.

When a compiler compiles (the definition of) a function with external linkage (in some other translation unit), the compiler writes the machine code code of that function at some address, and puts that address and the name of the function in the ".o" file where the linker can find it. The compiler assumes that the function will be called from some other translation unit (some other ".o" file), and must leave that function in this ".o" file, even if it ends up that the function is never called from any translation unit.

Most code conventions specify that header files contain only declarations, not definitions. Most code conventions specify that implementation files (".cpp" files) contain only definitions and local declarations, not external declarations.

This results in the "extern" keyword being used only in header files, never in implementation files. This results in internal linkage being indicated only in implementation files, never in header files. This results in the "static" keyword being used

only in implementation files, never in header files, except when "static" is used inside a class definition inside a header file, where it indicates something other than internal linkage.

We discuss header files and implementation files in more detail later in the FILE ORGANIZATION SECTION[175] of the book.

Internal

```
static
```

The **static** keyword can be used in four different ways:

- TO CREATE PERMANENT STORAGE FOR LOCAL VARIABLES IN A FUNCTION[176].
- TO SPECIFY INTERNAL LINKAGE[177].
- TO DECLARE MEMBER FUNCTIONS THAT ACT LIKE NON-MEMBER FUNCTIONS[178].
- TO CREATE A SINGLE COPY OF A DATA MEMBER[179].

Internal linkage

When used on a free function, a global variable, or a global constant, it specifies internal linkage (as opposed to `extern`, which specifies external linkage). Internal linkage limits access to the data or function to the current file.

Examples of use outside of any function or class:

```
static int apples = 15;
```

defines a "static global" variable named `apples`, with initial value 15, only visible from this translation unit.

```
static int bananas;
```

175 Chapter 3.1.5 on page 49
176 Chapter 3.3.4 on page 156
177 Chapter 3.2.4 on page 119
178 Chapter 4.3.5 on page 415
179 Chapter 4.3.4 on page 406

defines a "static global" variable named `bananas`, with initial value 0, only visible from this translation unit.

```
int g_fruit;
```

defines a global variable named `g_fruit`, with initial value 0, visible from every translation unit. Such variables are often frowned on as poor style.

```
static const int muffins_per_pan=12;
```

defines is a variable named `muffins_per_pan`, visible only in this translation unit. The static keyword is redundant here.

```
const int hours_per_day=24;
```

defines a variable named `hours_per_day`, only visible in this translation unit. (This acts the same as `static const int hours_per_day=24;`).

```
static void f();
```

declares that there is a function `f` taking no arguments and with no return value defined in this translation unit. Such a forward declaration is often used when defining mutually recursive functions.

```
static void f(){;}
```

defines the function `f()` declared above. This function can only be called from other functions and members in this translation unit; it is invisible to other translation units.

External

All entities in the C++ Standard Library have external linkage.

`extern`

The `extern` keyword tells the compiler that a variable is declared in another source module (outside of the current scope). The linker then finds this actual declaration and sets up the `extern` variable to point to the correct location. Variables described by `extern` statements will not have any space allocated for them, as they should be properly defined elsewhere. If a variable is declared extern, and the linker finds no actual declaration of it, it will throw an "Unresolved external symbol" error.

Examples:

```
extern int i;
```

declares that there is a variable named `i of type int`, defined somewhere in the program.

```
extern int j = 0;
```

defines a variable `j` with external linkage; the `extern` keyword is redundant here.

```
extern void f();
```

declares that there is a function `f` taking no arguments and with no return value defined somewhere in the program; `extern` is redundant, but sometimes considered good style.

```
extern void f() {;}
```

defines the function `f()` declared above; again, the `extern` keyword is technically redundant here as external linkage is default.

```
extern const int k = 1;
```

defines a constant `int k` with value `1` and external linkage; `extern` is required because `const` variables have internal linkage by default.

`extern` statements are frequently used to allow data to span the scope of multiple files.

When applied to function declarations, the additional "C" or "C++" string literal will change name mangling when compiling under the opposite language. That is, `extern "C" int plain_c_func(int param);` allows C++ code to execute a C library function plain_c_func.

3.3 Variables

Much like a person has a name that distinguishes him or her from other people, a *variable* assigns a particular instance of an object type, a *name* or *label* by which the instance can be referred to. The variable is the most important concept in programming, it is how the code can manipulate data. Depending on its use in the code a variable has a specific locality in relation to the hardware and based on the structure of the code it also has a specific scope where the compiler will recognize it as valid. All these characteristics are defined by a programmer.

3.3.1 Internal storage

We need a way to store data that can be stored, accessed and altered on the hardware by programming. Most computer systems operate using binary logic. The computer represents value using two voltage levels, usually 0V for *logic 0* and either +3.3 V or +5V for *logic 1*. These two voltage levels represent exactly two different values and by convention the values are zero and one. These two values, coincidentally, correspond to the two digits used by the binary number system. Since there is a correspondence between the logic levels used by the computer and the two digits used in the binary numbering system, it should come as no surprise that computers employ the binary system.

The Binary Number System

The binary number system uses base 2 which requires therefore only the digits *0* and *1*.

Bits and bytes

We typically write binary numbers as a sequence of bits (bits is short for binary digits). It is also a normal convention that these bit sequences, to make binary numbers more easier to read and comprehend, be added spaces in a specific relevant boundary, to be selected from the context that the number is being used. Much like we use a comma (UK and most ex-colonies) or a point to separated every three digits in larger decimal numbers. For example, the binary value 1010111110110010 could be written **1010 1111 1011 0010**.

These are defined boundaries for specific bit sequences.

Name	**Size (bits)**	**Example**
Bit	1	1
Nibble	4	0101
Byte	8	0000 0101
Word	16	0000 0000 0000 0101
Double Word	32	0000 0000 0000 0000 0000 0000 0000 0101

The bit

The smallest unit of data on a binary computer is a single bit. Since a single bit is capable of representing only two different values (typically zero or one) you may get the impression that there are a very small number of items you can represent with a single bit. Not true! There are an infinite number of items you can represent with a single bit.

With a single bit, you can represent any two distinct items. Examples include zero or one, true or false, on or off, male or female, and right or wrong. However, by using more than one bit, you will not be limited to representing binary data types (that is, those objects which have only two distinct values).

To confuse things even more, different bits can represent different things. For example, one bit might be used to represent the values zero and one, while an adjacent bit might be used to represent the colors red or black. How can you tell by looking at the bits? The answer, of course, is that you can't. But this illustrates the whole idea behind computer data structures: data is what you define it to be.

If you use a bit to represent a boolean (true/false) value then that bit (by your definition) represents true or false. For the bit to have any true meaning, you must be consistent. That is, if you're using a bit to represent true or false at one point in your program, you shouldn't use the true/false value stored in that bit to represent red or black later.

Since most items you will be trying to model require more than two different values, single bit values aren't the most popular data type. However, since everything else consists of groups of bits, bits will play an important role in your programs. Of course, there are several data types that require two distinct values, so it would seem that bits are important by themselves. however, you will soon see that individual bits are difficult to manipulate, so we'll often use other data types to represent boolean values.

The nibble

A nibble is a collection of bits on a 4-bit boundary. It would not be a particularly interesting data structure except for two items: BCD (binary coded decimal) numbers and hexadecimal (base 16) numbers. It takes four bits to represent a single BCD or hexadecimal digit.

With a nibble, we can represent up to 16 distinct values. In the case of hexadecimal numbers, the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are represented with four bits.

BCD uses ten different digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) and requires four bits. In fact, any sixteen distinct values can be represented with a nibble, but hexadecimal and BCD digits are the primary items we can represent with a single nibble.

The byte

The byte is the smallest individual piece of data that we can access or modify on a computer, it is without question, the most important data structure used by microprocessors today. Main memory and I/O addresses in the PC are all byte addresses.

A byte consists of eight bits and is the smallest addressable datum (data item) in the microprocessor, this is why processors only works on bytes or groups of bytes, never on bits. To access anything smaller requires that you read the byte containing the data and mask out the unwanted bits.

Since the computer is a byte addressable machine, it turns out to be more efficient to manipulate a whole byte than an individual bit or nibble. For this reason, most programmers use a whole byte to represent data types that require no more than 256 items, even if fewer than eight bits would suffice. For example, we will often represent the boolean values true and false by 00000001 and 00000000 (respectively).

Note:
This is why the ASCII CODE[a], is used in in most computers, it is based in a 7-bit non-weighted binary code, that takes advantage of the byte boundary.

a Chapter 4.8.1 on page 452

Probably the most important use for a byte is holding a character code. Characters typed at the keyboard, displayed on the screen, and printed on the printer all have numeric values.

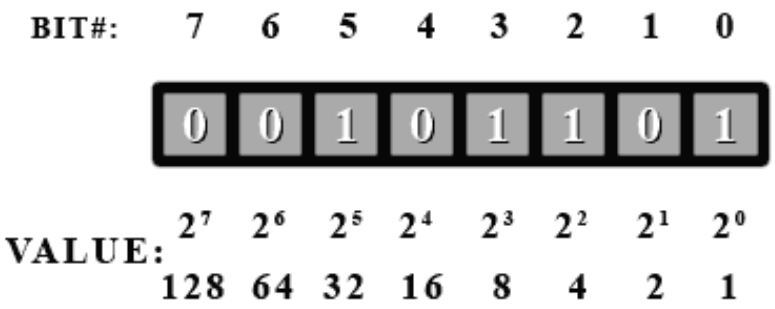


Figure 8: A byte contains 8 bits

A byte (usually) contains 8 bits. A bit can only have the value of 0 or 1. If all bits are set to 1, 11111111 in binary equals to 255 decimal.

The bits in a byte are numbered from bit zero (b0) through seven (b7) as follows: b7 b6 b5 b4 b3 b2 b1 b0

Bit 0 (b0) is the **low order bit** or **least significant bit** (**lsb**), bit 7 is the **high order bit** or **most significant bit** (**msb**) of the byte. We'll refer to all other bits by their number.

A byte also contains exactly two *nibbles*. Bits b0 through b3 comprise the low order nibble, and bits b4 through b7 form the high order nibble.

Since a byte contains eight bits, exactly two nibbles, byte values require two hexadecimal digits. It can represent 2^8, or 256, different values. Generally, we'll use a byte to represent:

1. unsigned numeric values in the range 0 => 255
2. signed numbers in the range -128 => +127
3. ASCII character codes
4. other special data types requiring no more than 256 different values. Many data types have fewer than 256 items so eight bits is usually sufficient.

In this representation of a computer byte, a bit number is used to label each bit in the byte. The bits are labeled from 7 to 0 instead of 0 to 7 or even 1 to 8, because processors always start counting at 0. It is simply more convenient to use 0 for computers as we shall see. The bits are also shown in descending order because, like with decimal numbers (normal base 10), we put the more significant digits to the left.

Consider the number 254 in decimal. The 2 here is more significant than the other digits because it represents hundreds as opposed to tens for the 5 or singles for the 4. The same is done in binary. The more significant digits are put towards the left. In binary, there are only 2 digits, instead of counting from 0 to 9, we only count from 0 to 1, but counting is done by exactly the same principles as counting in decimal. If we want to count higher than 1, then we need to add a more significant digit to the left. In decimal, when we count beyond 9, we need to add a 1 to the next significant digit. It sometimes may look confusing or different only because humans are used to counting with 10 digits.

Note:
The most significant digit in a byte is bit#7 and the least significant digit is bit#0. These are otherwise known as "msb" and "lsb" respectively in lowercase. If written in uppercase, MSB will mean most significant BYTE. You will see these terms often in programming or hardware manuals. Also, lsb is always bit#0, but msb can vary depending on how many bytes we use to represent numbers. However, we won't look into that right now.

In decimal, each digit represents multiple of a power of 10. So, in the decimal number 254.

- The *4* represents four multiples of one (4×10^0 since $10^0 = 1$).
- Since we're working in decimal (base 10), the *5* represents five multiples of 10 (5×10^1)
- Finally the *2* represents two multiples of 100 (2×10^2)

All this is elementary. The key point to recognize is that as we move from right to left in the number, the significance of the digits increases by a multiple of 10. This should be obvious when we look at the following equation:

$$(2 \times 10^2) + (5 \times 10^1) + (4 \times 10^0) = 254$$

In binary, each digit can only be one of two possibilities (0 or 1), therefore when we work with binary we work in base 2 instead of base 10. So, to convert the binary number 1101 to decimal we can use the following base 10 equation, which is very much like the one above:

$$(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 8 + 4 + 0 + 1 = 13$$

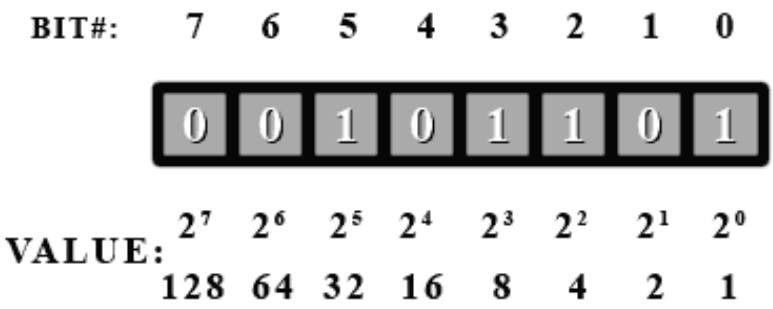


Figure 9: A byte contains 8 bits

To convert the number we simply add the bit values (2^n) where a 1 shows up. Let's take a look at our example byte again, and try to find its value in decimal.

First off, we see that bit #5 is a 1, so we have $2^5 = 32$ in our total. Next we have bit#3, so we add $2^3 = 8$. This gives us 40. Then next is bit#2, so 40 + 4 is 44. And finally is bit#0 to give 44 + 1 = 45. So this binary number is 45 in decimal.

As can be seen, it is impossible for different bit combinations to give the same decimal value. Here is a quick example to show the relationship between counting in binary (base 2) and counting in decimal (base 10).

$00_2 = 0_{10}$, $01_2 = 1_{10}$, $10_2 = 2_{10}$, $11_2 = 3_{10}$

The bases that these numbers are in are shown in subscript to the right of the number.

Carry bit

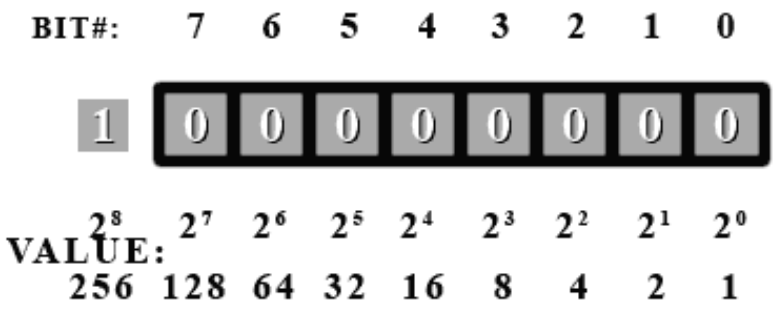


Figure 10

As a side note. What would happen if you added 1 to 255? No combination will represent 256 unless we add more bits. The next value (if we could have another digit) would be 256. So our byte would look like this.

But this 9^{th} bit (bit#8) doesn't exist. So where does it go? To be precise it actually goes into the carry bit. The carry bit resides in the processor of the computer, has an internal bit used exclusively for carry operations such as this. So if one adds 1 to 255 stored in a byte, the result would be 0 with the carry bit set in the CPU. Of course, a C++ programmer, never gets to use this bit directly. You'll would need to learn assembly to do that.

Endianness

After examining a single byte, it is time to look at ways to represent numbers larger than 255. This is done by grouping bytes together, we can represent numbers that are much larger than 255. If we use 2 bytes together, we double the number of bits in our number. In effect, 16 bits allows the representation numbers up to 65535 (`unsigned`), and 32 bits allows the representation of numbers above 4 billion.

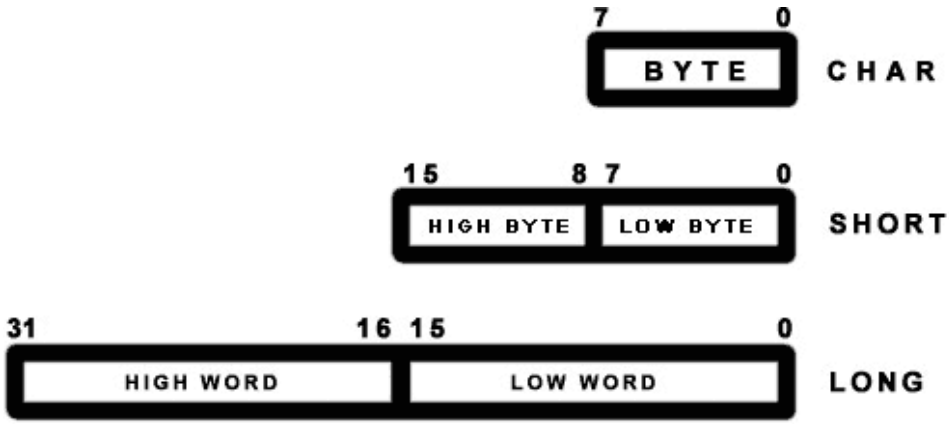


Figure 11: 3 basic primitive types char,short int,long int.

Here are a few basic primitive types:

- char (1 byte (by definition), max `unsigned` value: at least 255)
- short int (at least 16 bits, max `unsigned` value: at least 65535)
- long int (at least 32 bits, max `unsigned` value: at least 4294967295)
- float (typically 4 bytes, floating point)
- double (typically 8 bytes, floating point)

Note:
When using 'short int' and 'long int', you can leave out the 'int' as the compiler will know what type you want. You can also use 'int' by itself and it will default to whatever your compiler is set at for an int. On most recent compilers, int defaults to a 32-bit type.

All the information already given about the byte is valid for the other primitive types. The difference is simply the number of bits used is different and the msb is now bit#15 for a short and bit#31 for a long (assuming a 32-bit long type).

In a short (16-bit), one may think that in memory the byte for bits 15 to 8 would be followed by the byte for bits 7 to 0. In other words, byte #0 would be the high byte and byte #1 would be the low byte. This is true for some other systems. For example, the Motorola 68000 series CPUs do use this byte ordering. However, on PCs (with 8088/286/386/486/Pentiums) this is not so. The ordering is reversed so that the low byte comes before the high byte. The byte that represents bits 0 to 7

always comes before all other bytes on PCs. This is called little-endian ordering. The other ordering, such as on the M68000, is called big-endian ordering. This is very important to remember when doing low level byte operations that aim to be portable across systems.

For big-endian computers, the basic idea is to keep the higher bits on the left or in front. For little-endian computers, the idea is to keep the low bits in the low byte. There is no inherent advantage to either scheme except perhaps for an oddity. Using a little-endian long int as a smaller type of int is theoretically possible as the low byte(s) is/are always in the same location (first byte). With big-endian the low byte is always located differently depending on the size of the type. For example (in big-endian), the low byte is the 4^{th} byte in a long int and the 2^{nd} byte in a short int. So a proper cast must be done and low level tricks become rather dangerous.

To convert from one endianness to the other, one reverses the values of the bytes, putting the highest bytes value in the lowest byte and the lowest bytes value in the highest byte, and swap all the values for the in between bytes, so that if you had a 4 byte little-endian integer 0x0A0B0C0D (the 0x signifies that the value is hexadecimal) then converting it to big-endian would change it to 0x0D0C0B0A.

Bit endianness, where the bit order inside the bytes changes, is rarely used in data storage and only really ever matters in serial communication links, where the hardware deals with it.

Understanding two's complement

Two's complement is a way to store negative numbers in a pure binary representation. The reason that the two's complement method of storing negative numbers was chosen is because this allows the CPU to use the same add and subtract instructions on both signed and `unsigned` numbers.

To convert a positive number into its negative two's complement format, you begin by flipping all the bits in the number (1's become 0's and 0's become 1's) and then add 1. (This also works to turn a negative number back into a positive number Ex: -34 into 34 or vice-versa).

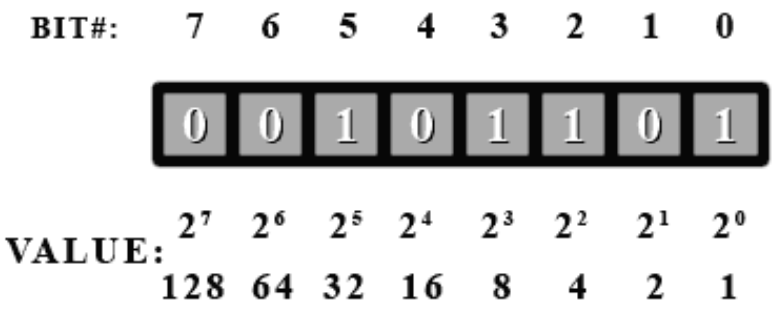


Figure 12: A byte contains 8 bits

Let's try to convert our number 45.

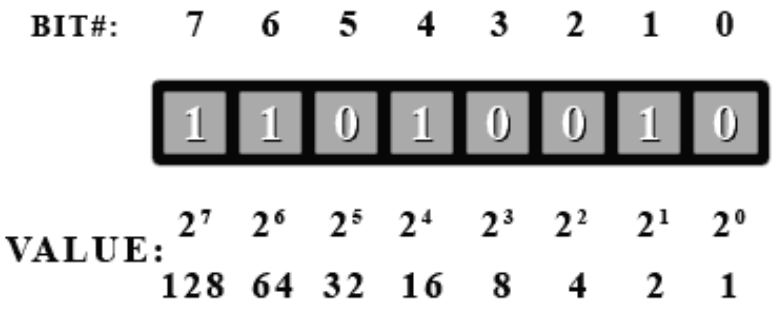


Figure 13: A byte contains 8 bits

First, we flip all the bits...

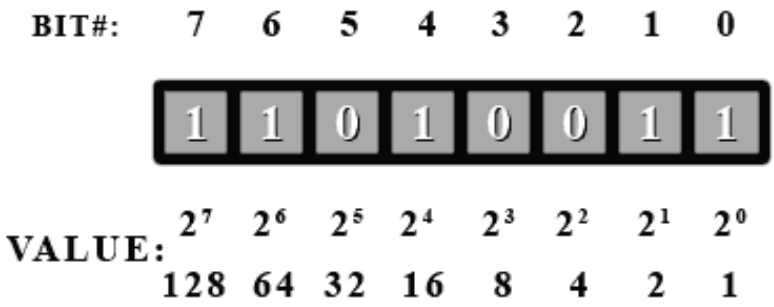


Figure 14: A byte contains 8 bits

And add 1. Now if we add up the values for all the one bits, we get... 128+64+16+2+1=211? What happened here? Well, this number actually is 211. It all depends on how you interpret it. If you decide this number is `unsigned`, then it's value is 211. But if you decide it's signed, then it's value is -45. It is completely up to you how you treat the number.

If and only if you decide to treat it as a signed number, then look at the msb (most significant bit [bit#7]). **If it's a 1, then it's a negative number.** If it's a 0, then it's positive. In C++, using `unsigned` in front of a type will tell the compiler you want to use this variable as an `unsigned` number, otherwise it will be treated as signed number.

Now, if you see the msb is set, then you know it's negative. So convert it back to a positive number to find out it's real value using the process just described above.

Let's go through a few examples.

Treat the following number as an `unsigned` byte. What is it's value in decimal?

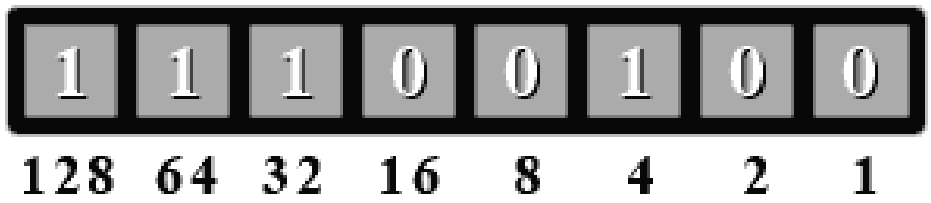


Figure 15: A byte contains 8 bits

Since this is an `unsigned` number, no special handling is needed. Just add up all the values where there's a 1 bit. 128+64+32+4=228. So this binary number is 228 in decimal.

Now treat the number above as a signed byte. What is its value in decimal?

Since this is now a signed number, we first have to check if the msb is set. Let's look. Yup, bit #7 is set. So we have to do a two's complement conversion to get its value as a positive number (then we'll add the negative sign afterwards).

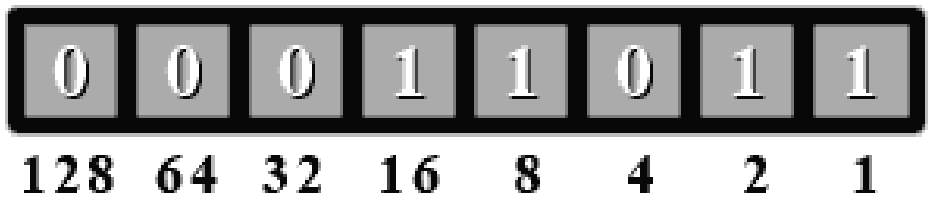


Figure 16: A byte contains 8 bits

Ok, so let's flip all the bits...

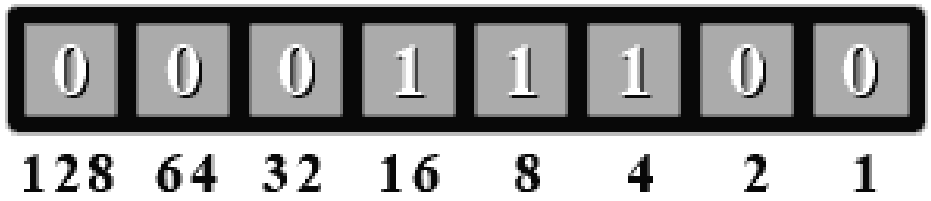


Figure 17: A byte contains 8 bits

And add 1. This is a little trickier since a carry propagates to the third bit. For bit#0, we do 1+1 = 10 in binary. So we have a 0 in bit#0. Now we have to add the carry to the second bit (bit#1). 1+1=10. bit#1 is 0 and again we carry a 1 over to the 3^{rd} bit (bit#2). 0+1 = 1 and we're done the conversion.

Now we add the values where there's a one bit. 16+8+4 = 28. Since we did a conversion, we add the negative sign to give a value of -28. So if we treat 11100100 (base 2) as a signed number, it has a value of -28. If we treat it as an `unsigned` number, it has a value of 228.

Let's try one last example.

Give the decimal value of the following binary number both as a signed and `unsigned` number.

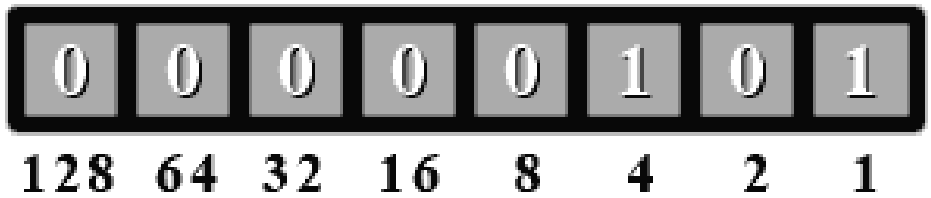


Figure 18: A byte contains 8 bits

First as an `unsigned` number. So we add the values where there's a 1 bit set. 4+1 = 5. For an `unsigned` number, it has a value of 5.

Now for a signed number. We check if the msb is set. Nope, bit #7 is 0. So for a signed number, it also has a value of 5.

As you can see, if a signed number doesn't have its msb set, then you treat it exactly like an `unsigned` number.

Note:
A special case of two's complement is where the sign bit (msb or bit#7 in a byte) is set to one and all other bits are zero, then its two's complement will be itself. It is a fact that two's complement notation (signed numbers) have 1 extra number than can be negative than positive. So for bytes, you have a range of -128 to +127. The reason for this is that the number zero uses a bit pattern (all zeros). Out of all the 256 possibilities, this leaves 255 to be split between positive and negative numbers. As you can see, this is an odd number and cannot be divided equally. If you were to try and split them, you would be left with the bit pattern described above where the sign bit is set (to 1) and all other bits are zeros. Since the sign bit is set, it has to be a negative number.
If you see this bit pattern of a sign bit set with everything else a zero, you cannot convert it to a positive number using two's complement conversion. The way you find out its value is to figure out the maximum number of bit patterns the value or type can hold. For a byte, this is 256 possibilities. Divide that number by 2 and put a negative sign in front. So -128 is this number for a byte. The following will be discussed below, but if you had 16 bits to work with, you have 65536 possibilities. Divide by 2 and add the negative sign gives a value of -32768.

Floating point representation

A generic real number with a decimal part can also be expressed in binary format. For instance 110.01 in binary corresponds to:

$$1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 2^2 + 2^1 + 2^{-2} = 6.25$$

Exponential notation (also known as scientific notation, or standard form, *when used with base 10*, as in 3×10^8) can be also used and the same number expressed as:

$$1.1001 \times 2^2 \qquad (= 11.001 \times 2^1 = 110.01)$$

When there is only one non-zero digit on the left of the decimal point, the notation is termed normalized.

In computing applications a real number is represented by a sign bit (S) an exponent (e) and a mantissa (M). The exponent field needs to represent both positive and negative exponents. To do this, a bias E is added to the actual exponent in order to get the stored exponent, and the sign bit (S), which indicates whether or not the number is negative, is transformed into either +1 or -1, giving s. A real number is thus represented as:

$$f = s \times M \times 2^{e-E}$$

S, e and M are concatenated one after the other in a 32-bit word to create a **single** precision floating point number and in a 64-bit doubleword to create a **double** precision one. For the single float type, 8 bits are used for the exponent and 23 bits for the mantissa, and the exponent offset is E=127. For the double type 11 bits are used for the exponent and 52 for the mantissa, and the exponent offset is E=1023.

There are two types of floating point numbers. *Normalized* and *denormalized*. A normalized number will have an exponent e in the range $0<e<2^8$ - 1 (between 00000000 and 11111111, non-inclusive) in a single precision float, and an exponent e in the range $0<e<2^{11}$ - 1 (between 00000000000 and 11111111111, non-inclusive) for a double float. Normalized numbers are represented as sign times 1.*Mantissa* times 2^{e-E}. Denormalized numbers are numbers where the exponent is 0. They are represented as sign times 0.*Mantissa* times 2^{1-E}. Denormalized numbers are used to store the value 0, where the exponent and mantissa are both 0. Floating point numbers can store both +0 and -0, depending on the sign. When the number isn't normalized or denormalized (it's exponent is all 1s) the number will be plus or minus infinity if the mantissa is zero and depending on the sign, or plus or minus NaN (Not a Number) if the mantissa isn't zero and depending on the sign.

For instance the binary representation of the number 5.0 (using float type) is:

0 10000001 01000000000000000000000

The first bit is 0, meaning the number is positive, the exponent is 129-127=2, and the mantissa is 1.01 (note the leading one is not included in the binary representation). 1.01 corresponds to 1.25 in decimal representation. Hence 1.25*4=5.

Floating point numbers are not always exact representations of values. a number like 10101101100011101010011101 couldn't be represented by a single precision floating point number because, disregarding the leading 1 which isn't part of the mantissa, there are 24 bits, and a single precision float can only store 23 numbers in its mantissa, so the 1 at the end would have to be dropped because it is the least significant bit. Also, there are some value which simply cannot be represented in binary which can be easily represented in decimal, E.g. 0.3 in decimal would be 0.0100110011001100110011... or something. A lot of other numbers cannot be exactly represented by a binary floating point number, no matter how many bits it use for it's mantissa, just because it would create a repeating pattern like this.

3.3.2 Locality (hardware)

Variables have two distinct characteristics: those that are created on the stack (local variables), and those that are accessed via a hard-coded memory address (global variables).

Globals

Typically a variable is bound to a particular address in COMPUTER MEMORY[180] that is automatically assigned to at runtime, with a fixed number of bytes determined by the size of the object type of a variable and any operations performed on the variable effects one or more VALUES[181] stored in that particular memory location.

All global defined variables will have static lifetime. Only those not defined as `const` will permit external linkage by default.

180 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPUTER%20MEMORY%20
181 HTTP://EN.WIKIPEDIA.ORG/WIKI/VALUE%20%28COMPUTER%20SCIENCE%29

Locals

If the size and location of a variable is unknown beforehand, the location in memory of that variable is stored in another variable instead, and the size of the original variable is determined by the size of the type of the second value storing the memory location of the first. This is called REFERENCING[182], and the variable holding the other variables memory location is called a pointer.

3.3.3 SCOPE[183]

Variables also reside in a specific SCOPE[184]. The scope of a variable is the most important factor to determines the life-time of a variable. Entrance into a scope begins the life of a variable and leaving scope ends the life of a variable. A variable is visible when in scope unless it is hidden by a variable with the same name inside an enclosed scope. A variable can be in global scope, `namespace` scope, file scope or compound statement scope.

As an example, in the following fragment of code, the variable 'i' is in scope only in the lines between the appropriate comments:

```
{
  int i; /*'i' is now in scope */
  i = 5;
  i = i + 1;
  cout << i;
}/* 'i' is now no longer in scope */
```

There are specific keywords that extend the life-time of a variable, and COMPOUND STATEMENT[185] define their own local SCOPE[186].

```
// Example of a compound statement defining a local scope
{

  {
    int i = 10; //inside a statement block
  }

 i = 2; //error, variable does not exist outside of the above compound statement
}
```

182 HTTP://EN.WIKIPEDIA.ORG/WIKI/REFERENCE%20%28COMPUTER%20SCIENCE%29
183 Chapter 3.1.9 on page 78
184 Chapter 3.1.9 on page 78
185 Chapter 3.1.7 on page 58
186 Chapter 3.1.9 on page 78

It is an error to declare the same variable twice within the same level of scope.

The only SCOPE[187] that can be defined for a global variable is a `namespace`, this deals with the visibility of variable not its validity, being the main purpose to avoid name collisions.

The concept of scope in relation to variables becomes extremely important when we get to classes, as the constructors are called when entering scope and the destructors are called when leaving scope.

Note:
Variables should be declared as local and as late as possible, and *initialized* immediately.

3.3.4 Type

So far we explained that internally data is stored in a way the hardware can read as zeros and ones, bits. That data is conceptually divided and labeled in accordance to the number of bits in each set. We must explain that since data can be interpreted in a variety of sets according to established formats as to represent meaningful information. This ultimately required that the programmer is capable of differentiate to the compiler what is needed, this is done by using the different types.

A variable can refer to simple values like integers called a *primitive type* or to a set of values called a *composite type* that are made up of PRIMITIVE TYPES[188] and other COMPOSITE TYPES[189]. Types consist of a set of valid values and a set of valid operations which can be performed on these values. A variable must declare what type it is before it can be used in order to enforce value and operation safety and to know how much space is needed to store a value.

Major functions that type systems provide are:

- **Safety** - types make it impossible to code some operations which cannot be valid in a certain context. This mechanism effectively catches the majority of common mistakes made by programmers. For example, an expression `"Hello, Wikipedia"/1` is invalid because a STRING LITERAL[190] cannot be divided by

187 Chapter 3.1.9 on page 78
188 HTTP://EN.WIKIPEDIA.ORG/WIKI/PRIMITIVE%20TYPES
189 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPOSITE%20TYPES
190 HTTP://EN.WIKIPEDIA.ORG/WIKI/STRING%20LITERAL

an INTEGER[191] in the usual sense. As discussed below, strong typing offers more safety, but it does not necessarily guarantee complete safety (see TYPE-SAFETY[192] for more information).
- **Optimization** - static type checking might provide useful information to a compiler. For example, if a type says a value is aligned at a multiple of 4, the memory access can be optimized.
- **Documentation** - using types in languages also improves DOCUMENTATION[193] of code. For example, the declaration of a variable as being of a specific type documents how the variable is used. In fact, many languages allow programmers to define semantic types derived from PRIMITIVE TYPE[194]s; either composed of elements of one or more primitive types, or simply as aliases for names of primitive types.
- **Abstraction** - types allow programmers to think about programs in higher level, not bothering with low-level implementation. For example, programmers can think of strings as values instead of a mere array of bytes.
- **Modularity** - types allow programmers to express the interface between two subsystems. This localizes the definitions required for interoperability of the subsystems and prevents inconsistencies when those subsystems communicate.

Data types

191 HTTP://EN.WIKIPEDIA.ORG/WIKI/INTEGER
192 HTTP://EN.WIKIPEDIA.ORG/WIKI/TYPE-SAFETY
193 HTTP://EN.WIKIPEDIA.ORG/WIKI/DOCUMENTATION
194 HTTP://EN.WIKIPEDIA.ORG/WIKI/PRIMITIVE%20TYPE

Type Primitive Types	Size in Bits	Alternate Names	
Type **Primitive Types**	**Size in Bits**	**Alternate Names**	
char	≥ 8	• `sizeof` gives the size in units of **char**s. These "BYTES[195]" need not be 8-bit bytes (though commonly they are); the number of bits is given by the `CHAR_BIT` macro in the `climits` header. • Signedness is implementation-defined. • Any encoding of 8 bits or less (e.g. ASCII) can be used to store characters. • Integer operations can be performed portably only for the range 0 ~ 127. • All bits contribute to the value of the **char**, i.e. there are no "holes" or "padding" bits.	—
signed char	same as **char**	• Characters stored like for type **char**. • Can store integers in the range -127 ~ 127 portably[1][196].	—
`unsigned` **char**	same as **char**	• Characters stored like for type **char**. • Can store integers in the range 0 ~ 255 portably.	—
short	≥ 16, ≥ size of **char**	• Can store integers in the range -32767 ~ 32767 portably[2][197]. • Used to reduce memory usage (although the resulting executable may be larger and probably slower as compared to using **int**.	**short int**, **signed short**, **signed short int**

195 HTTP://EN.WIKIPEDIA.ORG/WIKI/BYTE
196 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23TABLE%20OF%20TYPES%20FOOTNOTES
197 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23TABLE%20OF%20TYPES%20FOOTNOTES

Type	Size in Bits	Alternate Names	
Primitive Types			
`unsigned` **short**	same as **short**	• Can store integers in the range 0 ~ 65535 portably. • Used to reduce memory usage (although the resulting executable may be larger and probably slower as compared to using **int**.	`unsigned` **short int**
int	≥ 16, ≥ size of **short**	• Represents the "normal" size of data the processor deals with (the word-size); this is the integral data-type used normally. • Can store integers in the range -32767 ~ 32767 portably[2][198].	**signed**, **signed int**
`unsigned` **int**	same as **int**	• Can store integers in the range 0 ~ 65535 portably.	`unsigned`
long	≥ 32, ≥ size of **int**	• Can store integers in the range -2147483647 ~ 2147483647 portably[3][199].	**long int**, **signed long**, **signed long int**
`unsigned` **long**	same as **long**	• Can store integers in the range 0 ~ 4294967295 portably.	`unsigned` **long int**
bool	≥ size of **char**, ≤ size of **long**	• Can store the constants **true** and **false**.	—
wchar_t	≥ size of **char**, ≤ size of **long**	• Signedness is implementation-defined. • Can store "wide" (multi-byte) characters, which include those stored in a **char** and probably many more, depending on the implementation. • Integer operations are better not performed with **wchar_t**s. Use **int** or `unsigned` **int** instead.	—

198 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23TABLE%20OF%20TYPES%20FOOTNOTES
199 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23TABLE%20OF%20TYPES%20FOOTNOTES

Type	Size in Bits	Alternate Names	
Primitive Types			
float	≥ size of **char**	• Used to reduce memory usage when the values used do not vary widely. • The floating-point format used is implementation defined and need not be the IEEE single-precision format. • `unsigned` cannot be specified.	—
double	≥ size of **float**	• Represents the "normal" size of data the processor deals with; this is the floating-point data-type used normally. • The floating-point format used is implementation defined and need not be the IEEE double-precision format. • `unsigned` cannot be specified.	—
long double	≥ size of **double**	• `unsigned` cannot be specified.	—
User Defined Types			
struct or **class**	≥ sum of size of each member	• Default access modifier for **struct**s for members and base classes is **public**. • For **class**es the default is **private**. • The CONVENTION[200] is to use **struct** only for Plain Old Data types. • Said to be a *compound type*.	—
union	≥ size of the largest member	• Default access modifier for members and base classes is **public**. • Said to be a *compound type*.	—
enum	≥ size of **char**	• Enumerations are a distinct type from **int**s. `int`s are not implicitly converted to **enum**s, unlike in C. Also ++/-- cannot be applied to **enum**s unless overloaded.	—
`typedef`	same as the type being given a name	• Syntax similar to a storage class like **static**, `register` or **extern**.	—
template	≥ size of **char**	—	—

200 Chapter 3.1.7 on page 59

Type	Size in Bits	Alternate Names	
Primitive Types			
Derived Types[4][201]			
type`&` (reference)	≥ size of **char**	• References (unless optimized out) are usually internally implemented using pointers and hence they *do* occupy extra space separate from the locations they refer to.	—
type`*` (pointer)	≥ size of **char**	• `0` always represents the null pointer (an address where no data can be placed), irrespective of what bit sequence represents the value of a null pointer. • Pointers to different types may have different representations, which means they could also be of different sizes. So they are not convertible to one another. • Even in an implementation which guarantess all data pointers to be of the same size, function pointers and data pointers are in general incompatible with each other. • For functions taking variable number of arguments, the arguments passed must be of appropriate type, so even `0` must be cast to the appropriate type in such function-calls.	—
type `[`*integer*`]` (array)	≥ *integer* × size of *type*	• The brackets (`[]`) *follow* the identifier name in a declaration. • In a declaration which also initializes the array (including a function parameter declaration), the size of the array (the *integer*) can be omitted. • *type* `[]` is not the same as *type*`*`. Only under some circumstances one can be converted to the other.	—

201 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23TABLE%20OF%20TYPES%20FOOTNOTES

Type **Primitive Types**	Size in Bits	Alternate Names	
type `(`*comma-delimited list of types/declarations*`)` (function)	—	• The parentheses (`()`) *follow* the identifier name in a declaration, e.g. a 2-arg function pointer: **int** `(* fptr) (`**int** `arg1,` **int** `arg2)`. • Functions declared without any storage class are **extern**.	—
type aggregate_-type`::*` (member pointer)	≥ size of **char**	• `0` always represents the null pointer (a value which does not point to any member of the aggregate type), irrespective of what bit sequence represents the value of a null pointer. • Pointers to different types may have different representations, which means they could also be of different sizes. So they are not convertible to one another.	—

[1] -128 can be stored in two's-complement machines (i.e. most machines in existence).
[2] -32768 can be stored in two's-complement machines (i.e. most machines in existence).
[3] -2147483648 can be stored in two's-complement machines (i.e. most machines in existence).

[4] The precedences in a declaration are:	`[]`, `()` (left associative)	— Highest
	`&`, `*`, `::*` (right associative)	— Lowest

Note:
Many compilers also support the (non-standard) **long long** and `unsigned` **long long** data types. These can be expected to be added to the next revision of the C++ Standard (in fact, they are in the current draft for that standard, and have been standard in C since 1999).
Until the C++98 (and C99) standard adoption that defines **char** as signed, before the type was undefined in regard to the use of the sign. This information is important if you are using old compilers or reviewing old code.

Standard types

There are five basic *primitive types* called **standard types**, specified by particular keywords, that store a single value. These types stand isolated from the complexities of class type variables, even if the syntax of utilization at times brings them all in line, standard types do not share class properties (i.e.: don't have a constructor).

The type of a variable determines what kind of values it can store:

- `bool` - a boolean value: true; false
- `int` - Integer: -5; 10; 100
- `char` - a character in some encoding, often something like ASCII, ISO-8859-1 ("Latin 1") or ISO-8859-15: 'a', '=', 'G', '2'.
- `float` - floating-point number: 1.25; -2.35*10^23
- `double` - double-precision floating-point number: like float but more decimals

Note:
A `char` variable cannot store sequences of characters (strings), such as "C++" ({'C', '+', '+', '\0'}); it takes 4 `char` variables (including the null-terminator) to hold it. This is a common confusion for beginners. There are several types in C++ that store string values, but we will discuss them later.

The `float` and `double` primitive data types are called 'floating point' types and are used to represent real numbers (numbers with decimal places, like 1.435324 and 853.562). Floating point numbers and floating point arithmetic can be very tricky, due to the nature of how a computer calculates floating point numbers.

Note:
Don't use floating-point variables where discrete values are needed. Using a float for a loop counter is a great way to shoot yourself in the foot. Always test floating-point numbers as <= or >=, never use an exact comparison (== or !=).

Definition vs. declaration

There is an important concept, the distinction between the declaration of a variable and its definition, two separated steps involved in the use of variables. The declaration announces the properties (the type, size, etc.), on the other hand the definition causes storage to be allocated in accordance to the declaration.

Variables as function, classes and other constructs that require declarations may be declared many times, but each may only be defined one time.

Note:
There are ways around the definition limitation but uses and circumstances that may require it are vary rare or too specific that forgetting to interiorize the general rule is a quick way to get into errors that may be hard to resolve.

This concept will be further explained and with some particulars noted (such as `inline`) as we introduce other components. Here are some examples, some include concepts not yet introduced, but will give you a broader view:

```
   int an_integer;                                  // defines an_integer
   extern const int a = 1;                           // defines a
   int function( int b ) { return b+an_integer; }  // defines function and
defines b
   struct a_struct { int a; int b; };               // defines a_struct,
a_struct::a, and a_struct::b
   struct another_struct {                          // defines another_struct
     int a;                                        // defines nonstatic data
member a
     static int b;                                 // declares static data
member b
     another_struct(): a(0) { } };                 // defines a constructor of
another_struct
   int another_struct::b = 1;                      // defines another_struct::b
   enum { right, left };                            // defines right and left
   namespace FirstNamespace { int a; }             // defines FirstNamespace
and FirstNamespace::a
   namespace NextNamespace = FirstNamespace ;      // defines NextNamespace
   another_struct MySruct;                         // defines MySruct
   extern int b;                                    // declares b
   extern const int c;                              // declares c
   int another_function( int );                    // declares another_function
   struct aStruct;                                  // declares aStruct
   typedef int MyInt;                               // declares MyInt
   extern another_struct yet_another_struct;       // declares
yet_another_struct
   using NextNamespace::a;                          // declares NextNamespace::a
```

Declaration

C++ is a **statically typed** language. Hence, any variable cannot be used without specifying its type. This is why the type figures in the declaration. This way the compiler can protect you from trying to store a value of an incompatible type into a variable, e.g. storing a string in an integer variable. Declaring variables before use also allows spelling errors to be easily detected. Consider a variable

used in many statements, but misspelled in one of them. Without declarations, the compiler would silently assume that the misspelled variable actually refers to some other variable. With declarations, an "Undeclared Variable" error would be flagged. Another reason for specifying the type of the variable is so the compiler knows how much space in memory must be *allocated* for this variable.

The simplest variable declarations look like this (the parts in []s are optional):

[specifier(s)] type variable_name [= initial_value];

To create an integer variable for example, the syntax is

```
int sum;
```

where sum is the name you made up for the variable. This kind of statement is called a declaration. It *declares* `sum` as a variable of type **int**, so that `sum` can store an integer value. Every variable has to be declared before use and it is common practice to declare variables as close as possible to the moment where they are needed. This is unlike languages, such as C, where all declarations must precede all other statements and expressions.

In general, you will want to make up variable names that indicate what you plan to do with the variable. For example, if you saw these variable declarations:

```
char firstLetter;
char lastLetter;
int hour, minute;
```

you could probably make a good guess at what values would be stored in them. This example also demonstrates the syntax for declaring multiple variables with the same type in the same statement: hour and minute are both integers (*int* type). Notice how a comma separates the variable names.

```
int a = 123;
int b (456);
```

Those lines also declare variables, but this time the variables are *initialized* to some value. What this means is that not only is space allocated for the variables but the space is also filled with the given value. The two lines illustrate two different but equivalent ways to initialize a variable. The assignment operator '=' in a declaration has a subtle distinction in that it assigns an initial value instead of assigning a new value. The distinction becomes important especially when the values we are dealing with are not of simple types like integers but more complex objects like the input and output streams provided by the `iostream` class.

The expression used to initialize a variable need not be constant. So the lines:

```
int sum;
sum = a + b;
```

can be combined as:

```
int sum = a + b;
```

or:

```
int sum (a + b);
```

Declare a floating point variable 'f' with an initial value of 1.5:

```
float f = 1.5 ;
```

Floating point constants should always have a '.' (decimal point) somewhere in them. Any number that does not have a decimal point is interpreted as an integer, which then must be converted to a floating point value before it is used.

For example:

```
double a = 5 / 2;
```

will not set a to 2.5 because 5 and 2 are integers and integer arithmetic will apply for the division, cutting off the fractional part. A correct way to do this would be:

```
double a = 5.0 / 2.0;
```

You can also declare floating point values using scientific notation. The constant .05 in scientific notation would be 5×10^{-2}. The syntax for this is the base, followed by an e, followed by the exponent. For example, to use .05 as a scientific notation constant:

```
double a = 5e-2;
```

Note:
Single letters can sometimes be a bad choice for variable names when their purpose cannot be determined. However, some single-letter variable names are so commonly used that they're generally understood. For example `i`, `j`, and `k` are commonly used for loop variables and iterators; `n` is commonly used to represent the number of some elements or other counts; `s`, and `t` are commonly used for strings (that don't have any other meaning associated with them, as in utility routines); `c` and `d` are commonly used for characters; and `x` and `y` are commonly used for Cartesian co-ordinates.

Below is a program storing two values in integer variables, adding them and displaying the result:

```
// This program adds two numbers and prints their sum.
#include <iostream>

int main()
{
  int a = 123;
  int b (456);
  int sum;

  sum = a + b;

  std::cout << "The sum of " << a << " and " << b << " is " << sum << "\n";

  return 0;
}
```

6^{202}

or, if you like to save some space, the same above statement can be written as:

```
// This program adds two numbers and prints their sum, variation 1
#include <iostream>
#include <ostream>

using namespace std;

int main()
{
  int a = 123, b (456), sum = a + b;

  cout << "The sum of " << a << " and " << b << " is " << sum << endl;

  return 0;
}
```

202 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

203

`register`
The **register** keyword is a request to the compiler that the specified variable is to be stored in a register of the processor instead of memory as a way to gain speed, mostly because it will be heavily used. The compiler may ignore the request.

The keyword fell out of common use when compilers became better at most code optimizations than humans. Any valid program that uses the keyword will be semantically identical to one without it, unless they appear in a stringized macro (or similar context), where it can be useful to ensure that improper usage of the macro will cause a compile-time error. This keywords relates closely to `auto`.

```
register int x=99;
```

> **Note:**
> Register has different semantics between C and C++. In C it is possible to forbid the array-to-pointer conversion by making an array register declaration: `register int a[1];`.

Modifiers

There are several modifiers that can be applied to data types to change the range of numbers they can represent.

const

A variable declared with this specifier cannot be changed (as in read only). Either local or class-level variables (*scope*) may be declared `const` indicating that you don't intend to change their value after they're initialized. You declare a variable as being constant using the `const` keyword. Global `const` variables have static linkage. If you need to use a global constant across multiple files the best option is to use a special header file that can be included across the project.

```
const unsigned int DAYS_IN_WEEK = 7 ;
```

declares a positive integer constant, called `DAYS_IN_WEEK`, with the value 7. Because this value cannot be changed, you must give it a value when you declare it. If

203 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

you later try to assign another value to a constant variable, the compiler will print an error.

```
int main(){
  const int i = 10;

  i = 3;           // ERROR - we can't change "i"

  int &j = i;      // ERROR - we promised not to
                   // change "i" so we can't
                   // create a non-const reference
                   // to it

  const int &x = i; // fine - "x" is a const
                   // reference to "i"

  return 0;
}
```

The full meaning of `const` is more complicated than this; when working through pointers or references, `const` can be applied to mean that the object pointed (or referred) to will not be changed *via that pointer* or *reference*. There may be other names for the object, and it may still be changed using one of those names so long as it was not originally defined as being truly `const`.

It has an advantage for programmers over `#define` command because it is understood by the compiler, not just substituted into the program text by the preprocessor, so any error messages can be much more helpful.

With pointer it can get messy...

```
T const *p;                  // p is a pointer to a const T
T *const p;                  // p is a const pointer to T
T const *const p;            // p is a const pointer to a const T
```

If the pointer is a local, having a `const` pointer is useless. The order of T and const can be reversed:

```
const T *p;
```

is the same as

```
T const *p;
```

Note:
`const` can be used in the declaration of variables (arguments, return values and methods) - some of which we will mention later on.
Using `const` has several advantages:
To users of the `class`, it is immediately obvious that the `const` methods will not modify the object.
- Many accidental modifications of objects will be caught at compile time.
- Compilers like `const` since it allows them to do better optimization.

volatile
A hint to the compiler that a variable's value can be changed externally; therefore the compiler must avoid aggressive optimization on any code that uses the variable.

Unlike in Java, C++'s `volatile` specifier does not have any meaning in relation to multi-threading. Standard C++ does not include support for multi-threading (though it is a common extension) and so variables needing to be synchronized between threads need a synchronization mechanisms such as mutexes to be employed, keep in mind that `volatile` implies only safety in the presence of implicit or unpredictable actions by the same thread (or by a signal handler in the case of a **volatile sigatomic_t** object). Accesses to `mutable volatile` variables and fields are viewed as synchronization operations by most compilers and can affect control flow and thus determine whether or not other shared variables are accessed, this implies that in general ordinary memory operations cannot be reordered with respect to a mutable volatile access. This also means that mutable volatile accesses are sequentially consistent. This is not (as yet) part of the standard, it is under discussion and should be avoided until it gets defined.

mutable
This specifier may only be applied to a non-static, non-const member variables. It allows the variable to be modified within **const** member functions.

mutable is usually used when an object might be *logically constant*, i.e., no outside observable behavior changes, but not *bitwise* **const**, i.e. some internal member might change state.

The canonical example is the proxy pattern. Suppose you have created an image catalog application that shows all images in a long, scrolling list. This list could be modeled as:

```
class image {
```

```
 public:
   // construct an image by loading from disk
   image(const char* const filename);

   // get the image data
   char const * data() const;
 private:
   // The image data
   char* m_data;
}

class scrolling_images {
   image const* images[1000];
};
```

Note that for the image class, bitwise **const** and logically **const** is the same: If m_data changes, the public function data() returns different output.

At a given time, most of those images will not be shown, and might never be needed. To avoid having the user wait for a lot of data being loaded which might never be needed, the proxy pattern might be invoked:

```
class image_proxy {
  public:
   image_proxy( char const * const filename )
      : m_filename( filename ),
        m_image( 0 )
   {}
   ~image_proxy() { delete m_image; }
   char const * data() const {
      if ( !m_image ) {
         m_image = new image( m_filename );
      }
      return m_image->data();
   }
  private:
   char const* m_filename;
   mutable image* m_image;
};

class scrolling_images {
   image_proxy const* images[1000];
};
```

Note that the image_proxy does not change observable state when `data()` is invoked: it is logically constant. However, it is not bitwise constant since `m_image` changes the first time `data()` is invoked. This is made possible by declaring `m_image` mutable. If it had not been declared mutable, the `image_proxy::data()` would not compile, since `m_image` is assigned to within a constant function.

Note:
Like exceptions to most rules, the `mutable` keyword exists for a reason, but should not be overused. If you find that you have marked a significant number of the member variables in your class as `mutable` you should probably consider whether or not the design really makes sense.

short

The `short` specifier can be applied to the `int` data type. It can decrease the number of bytes used by the variable, which decreases the range of numbers that the variable can represent. Typically, a `short int` is half the size of a regular `int` -- but this will be different depending on the compiler and the system that you use. When you use the `short` specifier, the `int` type is implicit. For example:

```
short a;
```

is equivalent to:

```
short int a;
```

Note:
Although `short` variables may take up less memory, they can be slower than regular `int` types on some systems. Because most machines have plenty of memory today, it is rare that using a `short int` is advantageous.

long

The `long` specifier can be applied to the `int` and `double` data types. It can increase the number of bytes used by the variable, which increases the range of numbers that the variable can represent. A `long int` is typically twice the size of an `int`, and a `long double` can represent larger numbers more precisely. When you use `long` by itself, the `int` type is implied. For example:

```
long a;
```

is equivalent to:

```
long int a;
```

The shorter form, with the `int` implied rather than stated, is more idiomatic (i.e., seems more natural to experienced C++ programmers).

Use the `long` specifier when you need to store larger numbers in your variables. Be aware, however, that on some compilers and systems the long specifier may not increase the size of a variable. Indeed, most common 32-bit platforms (and one 64-bit platform) use 32 bits for `int` and also 32 bits for `long int`.

Note:
C++ does not yet allow `long long int` like modern C does, though it is likely to be added in a future C++ revision, and then would be guaranteed to be at least a 64-bit type. Most C++ implementations today offer `long long` or an equivalent as an *extension* to standard C++.

`unsigned`

The `unsigned` keyword is a data type specifier, that makes a variable only represent positive numbers and zero. It can be applied only to the `char`, `short`,`int` and `long` types. For example, if an `int` typically holds values from -32768 to 32767, an `unsigned int` will hold values from 0 to 65535. You can use this specifier when you know that your variable will never need to be negative. For example, if you declared a variable 'myHeight' to hold your height, you could make it unsigned because you know that you would never be negative inches tall.

Note:
unsigned types use MODULAR ARITHMETIC[a]. The default overflow behavior is to wrap around, instead of raising an exception or saturating. This can be useful, but can also be a source of bugs to the unwary.

a HTTP://EN.WIKIPEDIA.ORG/WIKI/MODULAR%20ARITHMETIC

signed

The `signed` specifier makes a variable represent both positive and negative numbers. It can be applied only to the `char`, `int` and `long` data types. The `signed` specifier is applied by default for `int` and `long`, so you typically will never use it in your code.

Note:
Plain `char` is a distinct type from both `signed char` and `unsigned char` although it has the same range and representation as one or the other. On some platforms plain `char` can hold negative values, on others it cannot. `char` should be used to represent a character; for a small integral type, use signed char, or for a small type supporting MODULAR ARITHMETIC[a] use `unsigned char`.

a HTTP://EN.WIKIPEDIA.ORG/WIKI/MODULAR%20ARITHMETIC

`static`

The **static** keyword can be used in four different ways:

- TO CREATE PERMANENT STORAGE FOR LOCAL VARIABLES IN A FUNCTION[204].
- TO SPECIFY INTERNAL LINKAGE[205].
- TO DECLARE MEMBER FUNCTIONS THAT ACT LIKE NON-MEMBER FUNCTIONS[206].
- TO CREATE A SINGLE COPY OF A DATA MEMBER[207].

Permanent storage

Using the `static` modifier makes a variable have static lifetime and on global variables makes them require internal linkage (variables will not be accessible from code of the same project that resides in other files).

static lifetime

Means that a static variable will need to be initialized in the file scope and at run time, will exist and maintain changes across until the program's process is closed, the particular order of destruction of static variables is undefined.

`static` variables instances share the same memory location. This means that they keep their value between function calls. For example, in the following code, a static variable inside a function is used to keep track of how many times that function has been called:

204 Chapter 3.3.4 on page 156
205 Chapter 3.2.4 on page 119
206 Chapter 4.3.5 on page 415
207 Chapter 4.3.4 on page 406

```
void foo() {
  static int counter = 0;
  cout << "foo has been called " << ++counter << " times\n";
}

int main() {
  for( int i = 0; i < 10; ++i ) foo();
}
```

Enumerated data type

In programming it is often necessary to deal with data types that describe a fixed set of alternatives. For example, when designing a program to play a card game it is necessary to keep track of the suit of an individual card.

One method for doing this may be to create unique constants to keep track of the suit. For example one could define

```
const int Clubs=0;
const int Diamonds=1;
const int Hearts=2;
const int Spades=3;

int current_card_suit=Diamonds;
```

Unfortunately there are several problems with this method. The most minor problem is that this can be a bit cumbersome to write. A more serious problem is that this data is indistinguishable from integers. It becomes very easy to start using the associated numbers instead of the suits themselves. Such as:

```
int current_card_suit=1;
```

...and worse to make mistakes that may be very difficult to catch such as a typo...

```
current_card_suit=11;
```

...which produces a valid expression in C++, but would be meaningless in representing the card's suit.

One way around these difficulty is to create a *new* data type specifically designed to keep track of the suit of the card, and *restricts* you to only use valid possibilities. We can accomplish this using an enumerated data type using the C++ `enum` keyword.

The `enum` keyword is used to create an enumerated type named name that consists of the elements in name-list. The var-list argument is optional, and can be used to create instances of the type along with the declaration.

Syntax

```
enum name {name-list} var-list;
```

For example, the following code creates the desired data type:

```
enum card_suit {Clubs,Diamonds,Hearts,Spades};
card_suit first_cards_suit=Diamonds;
card_suit second_cards_suit=Hearts;
card_suit third_cards_suit=0; //Would cause an error, 0 is an "integer" not a
 "card_suit"
card_suit forth_cards_suit=first_cards_suit; //OK, they both have the same type.
```

The line of code creates a new data type "`card_suit`" that may take on only one of four possible values: "`Clubs`", "`Diamonds`", "`Hearts`", and "`Spades`". In general the `enum` command takes the form:

```
enum new_type_name { possible_value_1,
                     possible_value_1,
                     /* ..., */
                     possible_value_n
} Optional_Variable_With_This_Type;
```

While the second line of code creates a new variable with this data type and initializes it to value to `Diamonds`". The other lines create new variables of this new type and show some initializations that are (and are not) possible.

Internally enumerated types are stored as integers, that begin with 0 and increment by 1 for each new possible value for the data type.

```
enum apples { Fuji, Macintosh, GrannySmith };
enum oranges { Blood, Navel, Persian };
apples pie_filling = Navel; //error can't make an apple pie with oranges.
apples my_fav_apple = Macintosh;
oranges my_fav_orange = Navel; //This has the same internal integer value as
 my_favorite_apple

//Many compilers will produce an error or warning letting you know your comparing
 two different quantities.
if(my_fav_apple == my_fav_orange)
  std::cout << "You shouldn't compare apples and oranges" << std::endl;
```

While enumerated types are not integers, they are in some case converted into integers. For example, when we try to send an enumerated type to standard output.

For example:

```
enum color {Red, Green, Blue};
color hair=Red;
color eyes=Blue;
```

```
color skin=Green;
std::cout << "My hair color is " << hair << std::endl;
std::cout << "My eye color is " << eyes << std::endl;
std::cout << "My skin color is " << skin << std::endl;
if (skin==Green)
  std::cout << "I am seasick!" << std::endl;
```

Will produce the output:

```
My hair color is 0

My eye color is 2

My skin color is 1

I am seasick!
```

We could improve this example by introducing an array that holds the names of our enumerated type such as:

```
std::string color_names[3]={"Red", "Green", "Blue"};
enum color {Red, Green, Blue};
color hair=Red;
color eyes=Blue;
color skin=Green;
std::cout << "My hair color is " << color_names[hair] << std::endl;
std::cout << "My eye color is " << color_names[eyes] << std::endl;
std::cout << "My skin color is " << color_names[skin] << std::endl;
```

In this case hair is automatically converted to an integer when it is index arrays. This technique is intimately tied to the fact that the color Red is internally stored as "0", Green is internally stored as "1", and Blue is internally stored as "2". **Be Careful!** One may override these default choices for the internal values of the enumerated types.

This is done by simply setting the value in the `enum` such as:

```
enum color {Red=2, Green=4, Blue=6};
```

In fact it is not necessary to an integer for every value of an enumerated type. In the case the value, the compiler will simply increase the value of the previous possible value by one.

Consider the following example:

```
enum colour {Red=2, Green, Blue=6, Orange};
```

Here the internal value of "`Red`" is 2, "`Green`" is 3, "`Blue`" is 6 and "`Orange` is 7. Be careful to keep in mind when using this that the internal values do not need to be unique.

Enumerated types are also automatically converted into integers in arithmetic expressions. Which makes it useful to be able to choose particular integers for the internal representations of an enumerated type.

One may have enumerated for the width and height of a standard computer screen. This may allow a program to do meaningful calculations, while still maintaining the benefits of an enumerated type.

```
enum screen_width {SMALL=800, MEDIUM=1280};
enum screen_height {SMALL=600, MEDIUM=768};
screen_width MyScreenW=SMALL;
screen_height MyScreenH=SMALL;
std::cout << "The number of pixels on my screen is " << MyScreenW*MyScreenH <<
 std::endl;
```

It should be noted that the internal values used in an enumerated type are constant, and cannot be changed during the execution of the program.

It is perhaps useful to notice that while the enumerated types can be converted to integers for the purpose arithmetic, they cannot be iterated through.

For example:

```
enum month { JANUARY=1, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST,
 SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER};

for( month cur_month = JANUARY; cur_month <= DECEMBER; cur_month=cur_month+1)
{
  std::cout << cur_month << std::endl;
}
```

This will fail to compile. The problem is with the `for` loop. The first two statements in the loop are fine. We may certainly create a new month variable and initialize it. We may also compare two months, where they will be compared as integers. We may **not** increment the cur_month variable. "`cur_month+1`" evaluates to an integer which may not be stored into a "`month`" data type.

In the code above we might try to fix this by **replacing** the `for` loop with:

```
for( int monthcount = JANUARY; monthcount <= DECEMBER; monthcount++)
{
  std::cout << monthcount  << std::endl;
}
```

This will work because we can increment the integer "`mounthcount`".

`typedef`

typedef keyword is used to give a data type a new alias.

```
typedef existing-type new-alias;
```

The intent is to make it easier the use of an awkwardly labeled data type, make external code conform to the coding styles or increase the comprehension of source code as you can use typedef to create a shorter, easier-to-use name for that data type. For example:

```
typedef int Apples;
typedef int Oranges;
Apples coxes;
Oranges jaffa;
```

The syntax above is a simplification. More generally, after the word "typedef", the syntax looks exactly like what you would do to declare a variable of the existing type with the variable name of the new type name. Therefore, for more complicated types, the new type name might be in the middle of the syntax for the existing type. For example:

```
typedef char (*pa)[3]; // "pa" is now a type for a pointer to an array of 3
chars
typedef int (*pf)(float); // "pf" is now a type for a pointer to a function
which
                          // takes 1 float argument and returns an int
```

This keyword also covered in the CODING STYLE CONVENTIONS SECTION[208].

Note:
You will only need to redeclare a typedef, if you want to redefine the same keyword.

Derived types

Type conversion

Type conversion or **typecasting** refers to changing an entity of one data type into another.

208 Chapter 3.1.8 on page 61

Implicit type conversion

Implicit type conversion, also known as **coercion**, is an automatic and temporary type conversion by the compiler. In a mixed-type expression, data of one or more subtypes can be converted to a supertype as needed at runtime so that the program will run correctly.

For example:

```
double  d;
long    l;
int     i;

if (d > i)      d = i;
if (i > l)      l = i;
if (d == l)     d *= 2;
```

As you can see d, l and i belong to different data types, the compiler will then automatically and temporarily converted the original types to equal data types each time a comparison or assignment is executed.

Note:
This behavior should be used with caution, and most modern compiler will provide a warning, as unintended consequences can arise.
Data can be lost when floating-point representations are converted to integral representations as the fractional components of the floating-point values will be truncated (rounded down). Conversely, converting from an integral representation to a floating-point one can also lose precision, since the floating-point type may be unable to represent the integer exactly (for example, float might be an IEEE 754 single precision type, which cannot represent the integer 16777217 exactly, while a 32-bit integer type can). This can lead to situations such as storing the same integer value into two variables of type int and type single which return false if compared for equality.

Explicit type conversion

Explicit type conversion manually converts one type into another, and is used in cases where automatic type casting doesn't occur.

```
double d = 1.0;

printf ("%d\n", (int)d);
```

In this example, *d* would normally be a double and would be passed to the PRINTF[209] function as such. This would result in unexpected behavior, since PRINTF[210] would try to look for an int. The typecast in the example corrects this, and passes the integer to PRINTF[211] as expected.

Note:
Explicit type casting should only be used as required. It should not be used if implicit type conversion would satisfy the requirements.

3.4 Operators

Now that we have covered the VARIABLES[212] and DATA TYPES[213] it becomes possible to introduce **operators**. **Operators** are special symbols that are used to represent and direct simple computations, this is significative importance in programming, since they serve to define, in a very direct, non-abtractive way and simple way, actions and simple interaction with data.

Since computers are mathematical devices, COMPILERS[214] and INTERPRETERS[215] require a full syntactic theory of all operations in order to parse formulas involving any combinations correctly. In particular they depend on OPERATOR PRECEDENCE[216] rules, on ORDER OF OPERATIONS[217], that are tacitly assumed in mathematical writing and the same applies to programming languages. Conventionally, the computing usage of *operator* also goes beyond the MATHEMATICAL USAGE[218] (for functions).

C++ like all PROGRAMMING LANGUAGES[219] uses a set of operators, they are subdivided into several groups:

- arithmetic operators (like addition and multiplication).

209 Chapter 3.7.11 on page 290
210 Chapter 3.7.11 on page 290
211 Chapter 3.7.11 on page 290
212 Chapter 3.2.4 on page 121
213 Chapter 3.3.3 on page 138
214 Chapter 3.1.10 on page 87
215 HTTP://EN.WIKIPEDIA.ORG/WIKI/INTERPRETER
216 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPERATOR%20PRECEDENCE
217 HTTP://EN.WIKIPEDIA.ORG/WIKI/ORDER%20OF%20OPERATIONS
218 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPERATOR
219 Chapter 2.1.3 on page 11

- boolean operators.
- string operators (used to manipulate STRINGS OF TEXT[220]).
- pointer operators.
- named operators (operators such as `sizeof`, `new`, and `delete` defined by alphanumeric names rather than a punctuation character).

Most of the operators in C++ do exactly what you would expect them to do, because most are common mathematical symbols. For example, the operator for adding two integers is +. C++ does allows the re-definition of some operators (OPERATOR OVERLOADING[221]) on more complex types, this be covered later on.

Expressions can contain both variables names and integer values. In each case the name of the variable is replaced with its value before the computation is performed.

3.4.1 Order of operations

When more than one operator appears in an expression the order of evaluation depends on the rules of precedence. A complete explanation of precedence can get complicated, but just to get you started:

Multiplication and division happen before addition and subtraction. So 2*3-1 yields 5, not 4, and 2/3-1 yields -1, not 1 (remember that in integer division 2/3 is 0). If the operators have the same precedence they are evaluated from left to right. So in the expression minute*100/60, the multiplication happens first, yielding 5900/60, which in turn yields 98. If the operations had gone from right to left, the result would be 59*1 which is 59, which is wrong. Any time you want to override the rules of precedence (or you are not sure what they are) you can use parentheses. Expressions in parentheses are evaluated first, so 2 * (3-1) is 4. You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even though it doesn't change the result.

3.4.2 PRECEDENCE[222] (Composition)

At this point we have looked at some of the elements of a programming language like variables, expressions, and statements in isolation, without talking about how to combine them.

220 HTTP://EN.WIKIPEDIA.ORG/WIKI/LITERAL%20STRING
221 Chapter 4.6 on page 438
222 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPERATOR%20PRECEDENCE

One of the most useful features of programming languages is their ability to take small building blocks and compose them (solving big problems by taking small steps at a time). For example, we know how to multiply integers and we know how to output values; it turns out we can do both at the same time:

```
std::cout << 17 * 3;
```

Actually, I shouldn't say "at the same time," since in reality the multiplication has to happen before the output, but the point is that any expression, involving numbers, characters, and variables, can be used inside an output statement. We've already seen one example:

```
std::cout << hour * 60 + minute << std::endl;
```

You can also put arbitrary expressions on the right-hand side of an assignment statement:

```
int percentage;
percentage = ( minute * 100 ) / 60;
```

This ability may not seem so impressive now, but we will see other examples where composition makes it possible to express complex computations neatly and concisely.

Note:
There are limits on where you can use certain expressions; most notably, the left-hand side of an assignment statement (normally) has to be a variable name, not an expression. That's because the left side indicates the storage location where the result will go. Expressions do not represent storage locations, only values.

The following is illegal:`minute+1 = hour;`

The exact rule for what can go on the left-hand side of an assignment expression is not so simple as it was in C; as OPERATOR OVERLOADING[223] and reference types can complicate the picture.

223 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPERATOR%20OVERLOADING

3.4.3 Chaining

```
std::cout << "The sum of " << a << " and " << b << " is " << sum
<< "\n";
```

The above line illustrates what is called *chaining of insertion operators* to print multiple expressions. How this works is as follows:

1. The leftmost insertion operator takes as its operands, `std::cout` and the string `"The sum of "`, it prints the latter using the former, and returns a *reference* to the former.
2. Now `std::cout << a` is evaluated. This prints the value contained in the location `a`, i.e. 123 and again returns `std::cout`.
3. This process continues. Thus, successively the expressions `std::cout << " and "`, `std::cout << b`, `std::cout << " is "`, `std::cout << " sum "`, `std::cout << "\n"` are evaluated and the whole series of chained values is printed.

224

3.4.4 Table of operators

Operators in the same group have the same **precedence** and the order of evaluation is decided by the **associativity** (*left-to-right* or *right-to-left*). Operators in a preceding group have *higher* precedence than those in a subsequent group.

Note:
Binding of operators actually cannot be completely described by "precedence" rules, and as such this table is an approximation. Correct understanding of the rules requires an understanding of the grammar of expressions.

Operators	Description	Example Usage	Associativity
Scope Resolution Operator			
`::`	unary scope resolution operator *for globals*	`::NUM_ELEMENTS`	—
`::`	binary scope resolution operator *for class and* `namespace` *members*	`std::cout`	

224 http://en.wikibooks.org/wiki/Category%3AC%2B%2B%20Programming

Function Call, Member Access, Post-Increment/Decrement Operators, RTTI and C++ Casts			
`()`	function call operator	`swap (x, y)`	
`[]`	array index operator	`arr [i]`	
`.`	member access operator *for an object of class/union type or a reference to it*	`obj.member`	Left to right
`->`	member access operator *for a pointer to an object of class/union type*	`ptr->member`	
`++ --`	post-increment/decrement operators	`num++`	
typeid `()`	run time type identification operator *for an object or type*	**typeid** `(std::cout)` **typeid** (std::iostream)	
`static_cast<>()` `dynamic_cast<>()` `const_cast<>()` `reinterpret_cast<>()`	C++ style cast operators for compile-time type conversion *See* TYPE CASTING[225] *for more info*	`static_cast<`**float**`>(i)` `dynamic_cast<std::istream>(stream)` `const_cast<`**char***`>("Hello, World!")` `reinterpret_cast<`**const long***`>("C++")`	
type `()`	functional cast operator *(*`static_cast`*is preferred for conversion to a primitive type)*	**float** `(i)`	
	also used as a constructor call for creating a temporary object, esp. of a class type	`std::string ("Hello, world!", 0, 5)`	
Unary Operators			
`!`, **not**	logical not operator	`!eof_reached`	
`~`, **compl**	bitwise not operator	`~mask`	
`+ -`	unary plus/minus operators	`-num`	
`++ --`	pre-increment/decrement operators	`++num`	Right to left
`&`, **bitand**	address-of operator	`&data`	
`*`	indirection operator	`*ptr`	

225 Chapter 3.4.14 on page 204

new **new**`[]` **new**`()` **new**`()[]`	new operators *for single objects or arrays*	**new** `std::string (5, '*')` **new** **int** `[100]` **new** `(raw_mem)` **int** **new** `(arg1, arg2)` **int** `[100]`	
delete **delete**`[]`	delete operator *for pointers to single objects or arrays*	**delete** `ptr` **delete**`[] arr`	
`sizeof` `sizeof()`	`sizeof` operator *for expressions or types*	`sizeof 123` `sizeof (`**int**`)`	
(*type*)	C-style cast operator *(deprecated)*	(**float**)`i`	
Member Pointer Operators			
`.*`	member pointer access operator *for an object of class/union type or a reference to it*	`obj.*memptr`	Right to left
`->*`	member pointer access operator *for a pointer to an object of class/union type*	`ptr->*memptr`	
Multiplicative Operators			
`* / %`	multiplication, division and modulus operators	`celsius_diff * 9 / 5`	Left to right
Additive Operators			
`+ -`	addition and subtraction operators	`end - start + 1`	Left to right
Bitwise Shift Operators			
`<<` `>>`	left and right shift operators	`bits << shift_len` `bits >> shift_len`	Left to right
Relational Inequality Operators			
`< > <= >=`	less-than, greater-than, less-than or equal-to, greater-than or equal-to	`i < num_elements`	Left to right
Relational Equality Operators			
`== !=`, **not_eq**	equal-to, not-equal-to	`choice != 'n'`	Left to right
Bitwise And Operator			Left to right

`&`, **bitand**		`bits & clear_mask_complement`	
Bitwise Xor Operator `^`, **xor**		`bits ^ invert_mask`	Left to right
Bitwise Or Operator `|`, **bitor**		bits | set_mask	Left to right
Logical And Operator `&&`, **and**		`arr != 0 && arr->len != 0`	Left to right
Logical Or Operator `||`, **or**		`arr == 0 || arr->len == 0`	Left to right
Conditional Operator `?:`		`size >= 0 ? size : 0`	Right to left
Assignment Operators `=` `+= -= *= /= %=` `&=`, **and_eq** `|=`, **or_eq** `^=`, **xor_eq** `<<= >>=`	assignment operator shorthand assignment operators (*foo* `op=` *barrepresents* *foo* `=` *foo* `op` *bar*)	`i = 0` `num /= 10`	Right to left
Exceptions **throw**		**throw** `"Array index out of bounds"`	—
Comma Operator `,`		`i = 0, j = i + 1, k = 0`	Left to right

226

226 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

3.4.5 Assignment

The most basic assignment operator is the "=" operator. It assigns one variable to have the value of another. For instance, the statement `x = 3` assigns `x` the value of 3, and `y = x` assigns whatever was in `x` to be in `y`. When the "=" operator is used to assign a class or struct, it acts like using the "=" operator on every single element. For instance:

```
//Example to demonstrate default "=" operator behavior.

struct A
 {
  int i;
  float f;
  A * next_a;
 };

//Inside some function
 {
  A a1, a2;             // Create two A objects.

  a1.i = 3;             // Assign 3 to i of a1.
  a1.f = 4.5;           // Assign the value of 4.5 to f in a1
  a1.next_a = &a2;      // a1.next_a now points to a2

  a2.next_a = NULL;     // a2.next_a is guaranteed to point at nothing now.
  a2.i = a1.i;          // Copy over a1.i, so that a2.i is now 3.
  a1.next_a = a2.next_a; // Now a1.next_a is NULL

  a2 = a1;              // Copy a2 to a1, so that now a2.f is 4.5. The other two
 are unchanged, since they were the same.
 }
```

Assignments can also be chained since the assignment operator returns the value it assigns. But this time the chaining is from right to left. For example, to assign the value of `z` to `y` and assign the same value (which is returned by the = operator) to `x` you use:

```
x = y = z;
```

When the "=" operator is used in a declaration, it has special meaning. It tells the COMPILER[227] to directly initialize the variable from whatever is on the right-hand side of the operator. This is called defining a variable, in the same way that you define a class or a function. With classes, this can make a difference, especially when assigning to a function call:

```
class A { /* ... */ };
```

227 Chapter 3.1.10 on page 87

```
A foo () { /* ... */ };

// In some function
 {
  A a;
  a = foo();

  A a2 = foo();
 }
```

In the first case, `a` is constructed, then is changed by the "=" operator. In the second statement, `a2` is constructed directly from the return value of `foo()`. In many cases, the COMPILER[228] can save a lot of time by constructing `foo()`'s return value directly into `a2`'s memory, which makes the program run faster.

Whether or not you define can also matter in a few cases where a definition can result in different linkage, making the variable more or less available to other source files.

3.4.6 Arithmetic operators

Arithmetic operations that can be performed on integers (also common in many other languages) include:

- Addition, using the + operator
- Subtraction, using the - operator
- Multiplication, using the * operator
- Division, using the / operator
- Remainder, using the % operator

Consider the next example, it will perform an addition and show the result:

```
#include<iostream>

using namespace std;
int main()
{
    int a = 3, b = 5;
    cout << a << '+' << b << '=' << (a+b);
    return 0;
}
```

The line relevant for the operatio is where the + operator adds the values stored in the locations `a` and `b`. `a` and `b` are said to be the *operands* of +. The combination

228 Chapter 3.1.10 on page 87

`a + b` is called an *expression*, specifically an *arithmetic expression* since + is an *arithmetic operator*.

Addition, subtraction and multiplication all do what you expect, but you might be surprised by division. For example, the following program:

```
int hour, minute;
hour = 11;
minute = 59;
std::cout << "Number of minutes since midnight: ";
std::cout << hour*60 + minute << std::endl;
std::cout << "Fraction of the hour that has passed: ";
std::cout << minute/60 << std::endl;
```

would generate the following output:

```
Number of minutes since midnight:  719


Fraction of the hour that has passed:  0
```

The first line is what we expected, but the second line is odd. The value of the variable minute is 59, and 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that C++ is performing integer division.

When both of the operands are integers (operands are the things operators operate on), the result must also be an integer, and by definition integer division always rounds down, even in cases like this where the next integer is so close.

A possible alternative in this case is to calculate a percentage rather than a fraction:

```
std::cout << "Percentage of the hour that has passed: ";
std::cout << minute*100/60 << std::endl;
```

The result is:

```
Percentage of the hour that has passed:  98
```

Again the result is rounded down, but at least now the answer is approximately correct. In order to get an even more accurate answer, we could use a different type of variable, called floating-point, that is capable of storing fractional values.

This next example:

```
#include<iostream>

using namespace std;
int main()
{
    int a = 33, b = 5;
```

```
    cout << "Quotient = " << a / b << endl;
    cout << "Remainder = "<< a % b << endl;
    return 0;
}
```

will return:

```
Quotient = 6
Remainder = 3
```

The *multiplicative* operators `*`, `/` and `%` are always evaluated before the *additive* operators `+` and `-`. Among operators of the same class, evaluation proceeds from left to right. This order can be overridden using grouping by parentheses, `(` and `)`; the expression contained within parentheses is evaluated before any other neighboring operator is evaluated. But note that some COMPILERS[229] may not strictly follow these rules when they try to optimize the code being generated, unless violating the rules would give a different answer.

For example the following statements convert a temperature expressed in degrees Celsius to degrees Fahrenheit and vice versa:

```
deg_f = deg_c * 9 / 5 + 32;
deg_c = ( deg_f - 32 ) * 5 / 9;
```

3.4.7 Compound assignment

One of the most common patterns in software with regards to operators is to update a value:

```
a = a + 1;
b = b * 2;
c = c / 4;
```

Since this pattern is used many times, there is a shorthand for it called compound assignment operators. They are a combination of an existing arithmetic operator and assignment operator:

- +=
- -=
- *=
- /=

229 Chapter 3.1.10 on page 87

- %=
- <<=
- >>=
- |=
- &=
- ^=

Thus the example given in the beginning of the section could be rewritten as

```
a += 1;  // Equivalent to (a = a + 1)
b *= 2;  // Equivalent to (b = b * 2)
c /= 4;  // Equivalent to (c = c / 4)
```

3.4.8 Character operators

Interestingly, the same mathematical operations that work on integers also work on characters.

```
char letter;
letter = 'a' + 1;
std::cout << letter << std::endl;
```

For the above example, outputs the letter b (on most systems -- note that C++ doesn't assume use of ASCII, EBCDIC, Unicode etc. but rather allows for all of these and other CHARSETS[230]). Although it is syntactically legal to multiply characters, it is almost never useful to do it.

Earlier I said that you can only assign integer values to integer variables and character values to character variables, but that is not completely true. In some cases, C++ converts automatically between types. For example, the following is legal.

```
int number;
number = 'a';
std::cout << number << std::endl;
```

On most mainstream desktop computers the result is 97, which is the number that is used internally by C++ on that system to represent the letter 'a'. However, it is generally a good idea to treat characters as characters, and integers as integers, and only convert from one to the other if there is a good reason. Unlike some other languages, C++ does not make strong assumptions about how the underlying platform represents characters; ASCII, EBCDIC and others are possible, and portable

230 HTTP://EN.WIKIPEDIA.ORG/WIKI/CHARSET

code will not make assumptions (except that '0', '1', ..., '9' are sequential, so that e.g. '9'-'0' == 9).

Automatic type conversion is an example of a common problem in designing a programming language, which is that there is a conflict between formalism, which is the requirement that formal languages should have simple rules with few exceptions, and convenience, which is the requirement that programming languages be easy to use in practice.

More often than not, convenience wins, which is usually good for expert programmers, who are spared from rigorous but unwieldy formalism, but bad for beginning programmers, who are often baffled by the complexity of the rules and the number of exceptions. In this book I have tried to simplify things by emphasizing the rules and omitting many of the exceptions.

3.4.9 Bitwise operators

These operators deal with a bitwise operations. Bit operations needs the understanding of binary numeration since it will deal with on one or two bit patterns or binary numerals at the level of their individual bits. On most microprocessors, bitwise operations are sometimes slightly faster than addition and subtraction operations and usually significantly faster than multiplication and division operations.

Bitwise operations especially important for much low-level programming from optimizations to writing device drivers, low-level graphics, communications protocol packet assembly and decoding.

Although machines often have efficient built-in instructions for performing arithmetic and logical operations, in fact all these operations can be performed just by combining the bitwise operators and zero-testing in various ways.

The bitwise operators work bit by bit on the operands. The operands must be of integral type (one of the types used for integers).

For this section, recall that a number starting with **0x** is hexadecimal (hexa, or hex for short or referred also as base-16). Unlike the normal decimal system using powers of 10 and the digits 0123456789, hex uses powers of 16 and the symbols 0123456789abcdef. In the examples remember that Oxc equals 1100 in binary and 12 in decimal. C++ does not directly support binary notation, which would hamper readability of the code.

NOT

~a

bitwise complement of **a**.

~0xc produces the value -1-0xc (in binary, ~1100 produces ...11110011 where "..." may be many more 1 bits)

The negation operator is a unary operator which precedes the operand, This operator must not be confused with the "logical not" operator, "!" (exclamation point), which treats the entire value as a single BOOLEAN[231]—changing a true value to false, and vice versa. The "logical not" is not a bitwise operation.

These others are binary operators which lie between the two operands. The precedence of these operators is lower than that of the relational and equivalence operators; it is often required to parenthesize expressions involving bitwise operators.

AND

a & b

bitwise boolean and of **a** and **b**

0xc & 0xa produces the value 0x8 (in binary, 1100 & 1010 produces 1000)

The TRUTH TABLE[232] of **a AND b**:

a	**b**	**^**
1	1	1
1	0	0
0	1	0
0	0	0

OR

a | b

bitwise boolean or of **a** and **b**

231 HTTP://EN.WIKIPEDIA.ORG/WIKI/BOOLEAN%20DATATYPE
232 HTTP://EN.WIKIPEDIA.ORG/WIKI/TRUTH%20TABLE

0xc | 0xa produces the value 0xe (in binary, 1100 | 1010 produces 1110)

The TRUTH TABLE[233] of **a OR b** is:

a	**b**	**∨**
1	1	1
1	0	1
0	1	1
0	0	0

XOR

a ^ b

bitwise xor of **a** and **b**

0xc ^ 0xa produces the value 0x6 (in binary, 1100 ^ 1010 produces 0110)

The TRUTH TABLE[234] of **a XOR b**:

a	**b**	**⊕**
1	1	0
1	0	1
0	1	1
0	0	0

Bit shifts

a << b

shift **a** left by **b** (multiply a by 2^b)

0xc << 1 produces the value 0x18 (in binary, 1100 << 1 produces the value 11000)

a >> b

233 HTTP://EN.WIKIPEDIA.ORG/WIKI/TRUTH%20TABLE
234 HTTP://EN.WIKIPEDIA.ORG/WIKI/TRUTH%20TABLE

shift **a** right by **b** (divide a by 2^b)

0xc >> 1 produces the value 0x6 (in binary, 1100 >> 1 produces the value 110)

3.4.10 Derived types operators

There are three data types known as pointers, references, and arrays, that have their own operators for dealing with them. Those are `*`, `&`, `[]`, `->`, `.*`, and `->*`.

Pointers, references, and arrays are fundamental data types that deal with accessing other variables. Pointers are used to pass around a variables *address* (where it is in memory), which can be used to have multiple ways to access a single variable. References are aliases to other objects, and are similar in use to pointers, but still very different. Arrays are large blocks of contiguous memory that can be used to store multiple objects of the same type, like a sequence of characters to make a string.

Subscript operator [ ]

This operator is used to access an object of an array. It is also used when declaring array types, allocating them, or deallocating them.

Arrays

An ARRAY[235] stores a constant-sized sequential set of blocks, each block containing a value of the selected type under a single name. Arrays often help organize collections of data efficiently and intuitively.

It is easiest to think of an *array* as simply a list with each value as an item of the list. Where individual elements are accessed by their position in the array called its index, also known as subscript. Each item in the array has an index from 0 to (the size of the array) -1, indicating its position in the array.

Advantages of arrays include:

- Random access in O(1) (BIG O NOTATION[236])
- Ease of use/port: Integrated into most modern languages

235 HTTP://EN.WIKIPEDIA.ORG/WIKI/ARRAY
236 HTTP://EN.WIKIPEDIA.ORG/WIKI/BIG%20O%20NOTATION

Disadvantages include:

- Constant size
- Constant data-type
- Large free sequential block to accommodate large arrays
- When used as non-static data members, the element type must allow default construction
- Arrays do not support copy assignment (you cannot write `arraya = arrayb`)
- Arrays cannot be used as the value type of a standard container
- Syntax of use differs from standard containers
- Arrays and inheritance don't mix (an array of Derived is not an array of Base, but can too easily be treated like one)

Note:
If complexity allows you should consider the use of containers (as in the C++ Standard Library). You should and can use for example `std::vector` which are as fast as arrays in most situations, can be dynamically resized, support iterators, and lets you treat the storage of the vector just like an array.
(Modern C allows VLAs, variable length arrays, but these are not used in C++, which already had a facility for re-sizable arrays in `std::vector`.)
The *pointer operator* as you will see is similar to the *array operator*.

For example, here is an array of integers, called List with 5 elements, numbered 0 to 4. Each element of the array is an integer. Like other integer variables, the elements of the array start out uninitialized. That means it is filled with unknown values until we initialize it by assigning something to it. (Remember primitive types in C are not initialized to 0.)

Index	**Data**
00	unspecified
01	unspecified
02	unspecified
03	unspecified
04	unspecified

Since an *array* stores values, what type of values and how many values to store must be defined as part of an array declaration, so it can allocate the needed space. The size of *array* must be a `const` integral expression greater than zero. That means that you cannot use user input to declare an *array*. You need to allocate the memory (with `operator new[]`), so the size of an array has to be known at compile time. Another disadvantage of the sequential storage method is that there

has to be a free sequential block large enough to hold the array. If you have an array of 500,000,000 blocks, each 1 byte long, you need to have roughly 500 megabytes of sequential space to be free; Sometimes this will require a defragmentation of the memory, which takes a long time.

To declare an array you can do:

```
int numbers[30]; // creates an array of 30 integers
```

or

```
char letters[4]; // create an array of 4 characters
```

and so on...

to initialize as you declare them you can use:

```
int vector[6]={0,0,1,0,0,0};
```

this will not only create the array with 6 int elements but also initialize them to the given values.

Assigning and accessing data

You can assign data to the array by using the name of the array, followed by the index.

For example to assign the number 200 into the element at index 2 in the array

```
List[2] = 200;
```

will give

Index	Data
00	unspecified
01	unspecified
02	200
03	unspecified
04	unspecified

You can access the data at an element of the array the same way.

```
std::cout << List[2] << std::endl;
```

This will print 200.

Basically working with individual elements in an array is no different then working with normal variables.

As you see accessing a value stored in an *array* is easy. Take this other example:

```
int x;
x = vector[2];
```

The above declaration will assign `x` the valued store at index 2 of variable `vector` which is 1.

Arrays are indexed starting at 0, as opposed to starting at 1. The first element of the array above is `vector[0]`. The index to the last value in the *array* is the *array* size minus one. In the example above the subscripts run from 0 through 5. C++ does not do bounds checking on array accesses. The compiler will not complain about the following:

```
char y;
int z = 9;
char vector[6] = { 1, 2, 3, 4, 5, 6 };

// examples of accessing outside the array. A compile error is not raised
y = vector[15];
y = vector[-4];
y = vector[z];
```

During program execution, an out of bounds *array* access does not always cause a run time error. Your program may happily continue after retrieving a value from `vector[-1]`. To alleviate indexing problems, the `sizeof` expression is commonly used when coding loops that process arrays.

```
int ix;
short anArray[]= { 3, 6, 9, 12, 15 };

for (ix=0; ix< (sizeof(anArray)/sizeof(short)); ++ix) {
  DoSomethingWith( anArray[ix] );
}
```

Notice in the above example, the size of the array was not explicitly specified. The compiler knows to size it at 5 because of the five values in the initializer list. Adding an additional value to the list will cause it to be sized to six, and because of the `sizeof` expression in the `for` loop, the code automatically adjusts to this change.

You can also use multi-dimensional arrays. The simplest type is a two dimensional array. This creates a rectangular array - each row has the same number of columns. To get a **char** array with 3 rows and 5 columns we write...

```
char two_d[3][5];
```

To access/modify a value in this array we need two subscripts:

```
char ch;
ch = two_d[2][4];
```

or

```
two_d[0][0] = 'x';
```

There are also weird notations possible:

```
int a[100];
int i = 0;
if (a[i]==i[a])
  printf("Hello World!\n");
```

`a[i]` and `i[a]` point to the same location. You will understand this better after knowing about pointers.

To get an `array` of a different size, you must explicitly deal with memory using `realloc`, `malloc`, `memcpy`, etc.

Why start at 0?
Most programming languages number arrays from 0. This is useful in languages where arrays are used interchangeably with a pointer to the first element of the array. In C++ the address of an element in the array can be computed from (address of first element) + i, where i is the index starting at 0 (a[1] == *(a + 1)). Notice here that "(address of the first element) + i" is not a literal addition of numbers. Different types of data have different sizes and the compiler will correctly take this into account. Therefore, it is simpler for the pointer arithmetic if the index started at 0.

Why no bounds checking on array indexes?
C++ does allow for, but doesn't force, bounds-checking implementations, in practice little or no checking is done. It affects storage requirements (needing "fat pointers") and impacts runtime performance. However, the `std::vector` template class as we will see is an object representing an array, and it provides the `at()` method, which does enforce bounds checking. Also in many implementations, the standard containers include particularly complete bounds checking in debug mode.

They might not support these checks in release builds, as *any* performance reduction in container classes relative to built-in arrays might prevent programmers from migrating from arrays to the more modern, safer container classes.

address-of operator &

To get the address of a variable so that you can assign a pointer, you use the "address of" operator, which is denoted by the ampersand `&` symbol. The "address of" operator does exactly what it says, it returns the "address of" a variable, a symbolic constant, or a element in an array, in the form of a pointer of the corresponding type. To use the "address of" operator, you tack it on in front of the variable that you wish to have the address of returned. It is also used when declaring reference types.

Now, do not confuse the "address of" operator with the declaration of a reference. Because use of operators is restricted to expression, the COMPILER[237] knows that `&sometype` is the "address of" operator being used to denote the return of the address of `sometype` as a POINTER[238].

References

References are a way of assigning a "handle" to a variable. References can also be thought of as "aliases"; they're not real objects, they're just alternative names for other objects.

Assigning References

This is the less often used variety of references, but still worth noting as an introduction to the use of references in function arguments. Here we create a reference that looks and acts like a standard variable except that it operates on the same data as the variable that it references.

```
int tZoo = 3;       // tZoo == 3
int &refZoo = tZoo; // tZoo == 3
refZoo = 5;         // tZoo == 5
```

`refZoo` is a reference to `tZoo`. Changing the value of `refZoo` also changes the value of `tZoo`.

237 Chapter 3.1.10 on page 87
238 Chapter 3.4.10 on page 184

Note:
One use of variable references is to pass function arguments using references. This allows the function to update / change the data in the variable being referenced

For example say we want to have a function to swap 2 integers

```
void swap(int &a, int &b){
  int temp = a;
  a = b;
  b = temp;
}


int main(){
   int x = 5;
   int y = 6;
   int &refx = x;
   int &refy = y;
   swap(refx, refy); // now x = 6 and y = 5
   swap(x, y); // and now x = 5 and y = 6 again
}
```

References cannot be null as they refer to instantiated objects, while pointers can be null. References cannot be reassigned, while pointers can be.

```
int main(){
   int x = 5;
   int y = 6;
   int &refx = x;
   &refx = y; // won't compile
}
```

As references provide strong guarantees when compared with pointers, using references makes the code simpler. Therefore using references should usually be preferred over using pointers. Of course, pointers have to be used at the time of dynamic memory allocation (new) and deallocation (delete).

Pointers, Operator *

The * operator is used when declaring pointer types but it is also used to get the variable pointed to by a pointer.

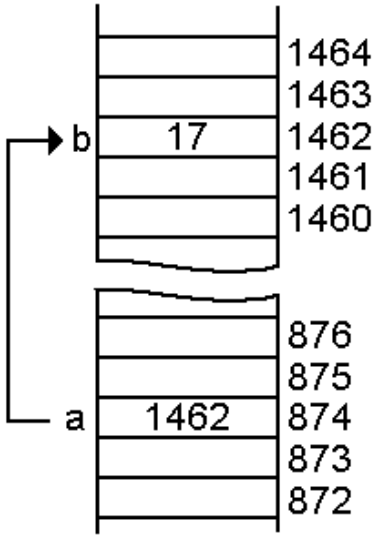


Figure 19: Pointer *a* pointing variable *b*. Note that *b* stores number, whereas *a* stores address of *b* in memory (1462)

Pointers are important data types due to special characteristics. They may be used to indicate a variable without actually creating a variable of that type. They can be a difficult concept to understand, some special effort should be spent on understanding the power they give to programmers.

Pointers have a very descriptive name. Pointers variables only store memory addresses, usually the addresses of other variables. Essentially, they point to another variable memory location, a reserved location on the computer memory. You can use a pointer to PASS THE LOCATION OF A VARIABLE TO A FUNCTION[239], this enables the function's pointer to use the variable space, so that it can retrieve or modify its data. You can even have pointers to pointers, and pointers to pointers to pointers and so on and so forth.

Declaring

Pointers are declared by adding a * before the variable name in the declaration, as in the following example:

239 Chapter 3.7 on page 229

```
int* x;  // pointer to int.
int * y; // pointer to int. (legal, but rarely used)
int *z;  // pointer to int.
int*i;   // pointer to int. (legal, but rarely used)
```

Note:
As always whitespace does not matter, so the position of the * doesn't matter only the order of the use.

Due to historical reasons some programmers refer to a specific use as:

```
// C codestyle int *z;
// C++ codestyle int* z;
```

As seen before on the CODING STYLE CONVENTIONS SECTION[a] adherence to a single style is preferred.

a Chapter 3.1.7 on page 59

Watch out, though, because the * associates to the following declaration only:

```
int* i, j;  // CAUTION! i is pointer to int, j is int.
int *i, *j; // i and j are both pointer to int.
```

You can also have multiple pointers chained together, as in the following example:

```
int **i;  // Pointer to pointer to int.
int ***i; // Pointer to pointer to pointer to int (rarely used).
```

Assigning values

Everyone gets confused about pointers as assigning values to pointers may be a bit tricky but if you know the basic you can proceed more easily. By carefully going through the examples rather than a simple description, try to understand the points as they are presented to you.

Assigning values to pointers (non-char type)

```
double vValue = 25.0;// declares and initializes a vValue as type double
double* pValue = &vValue;
cout << *pValue << endl;
```

The second statement uses "`&`" the reference operator and "`*`" to tell the compiler this is a pointer variable and assign `vValue` variable's address to it. In the last

statement, it outputs the value from the `vValue` variable by de-referencing the pointer using the `"*"` operator.

Assigning values to pointers (char type)

```
char pArray[20] = {"Name1"};
char* pValue(pArray);// or 0 in old compilers, nullptr is a part of C++0X
pValue = "Value1";
cout << pValue  << endl ;// this will return the Value1;
```

So as mentioned early, a pointer is a variable which stores the address of another variable, as you need to initialize an array because you can not directly assign values to it. You will need to use pointers directly or a pointer to array in a mixed context, to use pointers alone, examine the next example.

```
char* pValue("String1");
pValue = "String2";
cout << pValue << endl ;
```

Remember you can't leave the pointer alone or initialize it as nullptr cause it will case an error. The compiler thinks it is as a memory address holder variable since you didn't point to anything and will try to assign values to it, that will cause an error since it does not point to anywhere.

Dereferencing

This is the `*` operator. It is used to get the variable pointed to by a pointer. It is also used when declaring pointer types.

When you have a pointer, you need some way to access the memory that it points to. When it is put in front of a pointer, it gives the variable pointed to. This is an lvalue, so you can assign values to it, or even initialize a reference from it.

```
#include <iostream>

int main()
{
  int i;
  int * p = &i;
  i = 3;

  std::cout<<*p<<std::endl; // prints "3"

  return 0;
}
```

Since the result of an `&` operator is a pointer, `*&i` is valid, though it has absolutely no effect.

Now, when you combine the <Tt>* operator with classes, you may notice a problem. It has lower precedence than `.`! See the example:

```
struct A { int num; };

A a;
int i;
A * p;

p = &a;
a.num = 2;

i = *p.num; // Error! "p" isn't a class, so you can't use "."
i = (*p).num;
```

The error happens because the compiler looks at *p.num* first ("." has higher precedence than "*") and because *p* does not have a member named *num* the compiler gives you an error. Using grouping symbols to change the precedence gets around this problem.

It would be very time-consuming to have to write `(*p).num` a lot, especially when you have a lot of classes. Imagine writing `(*(*(*(*MyPointer).Member).SubMember).Value).WhatIWant`! As a result, a special operator, `->`, exists. Instead of `(*p).num`, you can write `p->num`, which is completely identical for all purposes. Now you can write `MyPointer->Member->SubMember->Value->WhatIWant`. It's a lot easier on the brain!

Null pointer

The null pointer is a special status of pointers. It means that the pointer points to absolutely nothing. It is an error to attempt to dereference (using the `*` or `->` operators) a null pointer. A null pointer can be referred to using the constant zero, as in the following example:

```
int i;
int *p;

p = 0; //Null pointer.
p = &i; //Not the null pointer.
```

Note that you can't assign a pointer to an integer, even if it's zero. It has to be the constant. The following code is an error:

```
int i = 0;
int *p = i; //Error: 0 only evaluates to null if it's a pointer
```

There is an old macro, defined in the standard library, derived from the C language that inconsistently has evolved into #define NULL ((void *)0), this makes `NULL`, always equal to a null pointer value (essentially, 0).

Note:
It is considered as good practice to avoid the use of macros and defines as much as possible. In the particular case at hand the NULL isn't type-safe. Any rational to use it for visibility of the use of a pointer can be addressed by the proper naming of the pointer variable.

Since a null pointer is 0, it will always compare to 0. Like an integer, if you use it in a true/false expression, it will return false if it is the null pointer, and true if it's anything else:

```
#include <iostream>

void IsNull (int * p)
{
  if (p)
    std::cout<<"Pointer is not NULL"<<std::endl;
  else
    std::cout<<"Pointer is NULL"<<std::endl;
}

int main()
{
  int * p;
  int i;

  p = NULL;
  IsNull(p);
  p = &i;
  IsNull(&i);
  IsNull(p);
  IsNull(NULL);

  return 0;
}
```

This program will output that the pointer is NULL, then that it isn't NULL twice, then again that it is.

Pointers and multi-dimensional arrays

Pointers and Multi-Dimensional non-Char Arrays

This is tricky part and might be hard but relatively than next part we are going to talk about ,first of all you need to know at least how to use Two Dimensional Arrays /Assign Values to Arrays / Return Values from Arrays ,since this is reserved for Pointer I am not going to mention about Arrays separately but when Arrays needed it will mixed up with pointer

The main objects are

1. Assign Values to Multi Dimensional Pointers
2. How to use Pointers with Multi Dimensional Arrays
3. Return Values
4. Initialize Pointers and Arrays
5. How to Arrange Values in them

1. **Assign Values to Multi Dimensional Pointers.**

In non-Char Type you need to involve arrays with Pointers cause since Pointers treat char* type to in special way and other type to another way like only refer the address or get the address and get the value by indirect method.

If you declare it like this way:

```
double (*pDVal)[2] = {{1,2},{1,2}};
```

It will probably generate an error! Because pointers used in non-Char type only directly, in char types refer the address of another variable by assigning a variable first then you can get its(that assigned variable)value by indirect way.!

```
double ArrayVal[5][5] = {
{1,2,3,4,5},
{1,2,3,4,5},
{1,2,3,4,5},
{1,2,3,4,5},
{1,2,3,4,5},
};

double(*pArray)[5] = ArrayVal;
*(*(pArray+0)+0) = 10;
*(*(pArray+0)+1) = 20;
*(*(pArray+0)+2) = 30;
*(*(pArray+0)+3) = 40;
*(*(pArray+0)+4) = 50;
*(*(pArray+1)+0) = 60;
*(*(pArray+1)+1) = 70;
*(*(pArray+1)+2) = 80;
*(*(pArray+1)+3) = 90;
*(*(pArray+1)+4) = 100;
*(*(pArray+2)+0) = 110;
*(*(pArray+2)+1) = 120;
```

```
*(*(pArray+2)+2) = 130;
*(*(pArray+2)+3) = 140;
*(*(pArray+2)+4) = 150;
*(*(pArray+3)+0) = 160;
*(*(pArray+3)+1) = 170;
*(*(pArray+3)+2) = 180;
*(*(pArray+3)+3) = 190;
*(*(pArray+3)+4) = 200;
*(*(pArray+4)+0) = 210;
*(*(pArray+4)+1) = 220;
*(*(pArray+4)+2) = 230;
*(*(pArray+4)+3) = 240;
*(*(pArray+4)+4) = 250;
```

There is another way instead

```
*(*(pArray+0)+0)
```

it's

```
*(pArray[0]+0)
```

You can use one of them to assign value to Array through the pointer to return values you can use either the appropriate Array or Pointer.

Pointers and multi-dimensional char arrays

This is bit hard and even hard to remember so I suggest keep practice until you get the spirit Pointers only.! You can't use Pointers + Multi Dimensional Arrays with Char Type. Only for non-char type.

Multi-dimensional pointer with char type

```
char* pVar[5] = { "Name1" , "Name2" , "Name3", "Name4", "Name5" }

pVar[0] = "XName01";
cout << pVar[0] << endl ; //this will return the XName01 instead Name1 which was
 replaced with Name1.
```

in here the 5 means of the first statement is the number of rows (there are no columns need to be specified in pointer it's only in Arrays) the next statement assigns another string to position 0 which is the position of first place of first statement. finally return the answer

Dynamic memory allocation

In your system memory each memory block got an address so whenever you compile the code at the beginning all variable reserve some space in the memory but in Dynamic Memory Allocation it only reserve when it needed it means at execution time of that statement this allocates memory in your free space area(unused space) so it means if there is no space or no contiguous blocks then the compiler will generate and error message

Dynamic memory allocation and pointer non-char type

This is same as assign non-char 1 dimensional Array to Pointer

```
double* pVal = new double[5];
//or double* pVal = new double; // this line leaves out the necessary memory
 allocation
*(pVal+0) = 10;
*(pVal+1) = 20;
*(pVal+2) = 30;
*(pVal+3) = 40;
*(pVal+4) = 50;

cout << *(pVal+0) << endl;
```

The first statement's Lside(left side) declares an variable and Rside request a space for double type variable and allocate it in free space area in your memory. So next and so fourth you can see it increases the integer value that means *(pVal+0) pVal -> if this uses alone it will return the address corresponding to first memory block. (that used to store the 10) and 0 means move 0 block ahead but it's 0 it means don't move stay in current memory block, and you use () parenthesis cause + < * < () consider the priority so you need to use parenthesis avoid to calculating the * fist

- is called INDIRECT Operator which DE-REFERENCE THE Pointer and return the value corresponding to the memory block.

(Memory Block Address+steps)

- -> De-reference.

Dynamic memory allocation and pointer char type

```
char* pVal = new char;
pVal = "Name1";
cout << pVal << endl;
delete pVal; //this will delete the allocated space
pVal = nullptr //null the pointer
```

You can see this is the same as static memory declaration, in static declaration it goes:

```
char* pVal("Name1");
```

Dynamic memory allocation and pointer non-char array type

```
double (*pVal2)[2]= new double[2][2]; //this will add 2x2 memory blocks to type
 double pointer
*(*(pVal2+0)+0) = 10;
*(*(pVal2+0)+1) = 10;
*(*(pVal2+0)+2) = 10;
*(*(pVal2+0)+3) = 10;
*(*(pVal2+0)+4) = 10;
*(*(pVal2+1)+0) = 10;
*(*(pVal2+1)+1) = 10;
*(*(pVal2+1)+2) = 10;
*(*(pVal2+1)+3) = 10;
*(*(pVal2+1)+4) = 10;
```

```
delete [] pVal; //it doesn't matter the dimension you only need to mention []
pVal = nullptr
```

Note:
Never use a multi-dimensional pointer array with char type, as it will generate an error.

```
char (*pVal)[5] ;// this is different from pointer of array
// which is char* pVal[5] ;
```

But both are different.

Pointers to classes

Indirection operator ->

This pointer indirection operator is used to access a member of a class pointer.

Member dereferencing operator .*

This pointer-to-member dereferencing operator is used to access the variable associated with a specific class instance, given an appropriate pointer.

Member indirection operator ->*

This pointer-to-member indirection operator is used to access the variable associated with a class instance pointed to by one pointer, given another pointer-to-member that's appropriate.

Pointers to functions

When used to point to functions, pointers can be exceptionally powerful. A call can be made to a function anywhere in the program, knowing only what kinds of parameters it takes. POINTERS TO FUNCTIONS[240] are used several times in the standard library, and provide a powerful system for other libraries which need to adapt to any sort of user code. This case is examined more in depth in the FUNCTIONS SECTION[241] of this book.

242

3.4.11 sizeof

The sizeof keyword refers to an operator that works at compile time to report on the size of the storage occupied by a TYPE[243] of the argument passed to it (equivalently, by a variable of that type). That size is returned as a multiple of the size of a char, which on many personal computers is 1 byte (or 8 bits). The number of bits in a char is stored in the CHAR_BIT constant defined in the <climits> header file. This is one of the operators for which OPERATOR OVERLOADING[244] is not allowed.

```
//Examples of sizeof use
int int_size( sizeof( int ) );// Might give 1, 2, 4, 8 or other values.

// or

int answer( 42 );
int answer_size( sizeof( answer ) );// Same value as sizeof( int )
int answer_size( sizeof answer);    // Equivalent syntax
```

For example, the following code uses sizeof to display the sizes of a number of variables:

```
struct EmployeeRecord {
  int ID;
```

240 Chapter 3.7.7 on page 255
241 Chapter 3.7 on page 229
242 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
243 Chapter 3.3.3 on page 138
244 Chapter 4.6 on page 438

```
  int age;
  double salary;
  EmployeeRecord* boss;
};

//...

cout << "sizeof(int): " << sizeof(int) << endl
     << "sizeof(float): " << sizeof(float) << endl
     << "sizeof(double): " << sizeof(double) << endl
     << "sizeof(char): " << sizeof(char) << endl
     << "sizeof(EmployeeRecord): " << sizeof(EmployeeRecord) << endl;

int i;
float f;
double d;
char c;
EmployeeRecord er;

cout << "sizeof(i): " << sizeof(i) << endl
     << "sizeof(f): " << sizeof(f) << endl
     << "sizeof(d): " << sizeof(d) << endl
     << "sizeof(c): " << sizeof(c) << endl
     << "sizeof(er): " << sizeof(er) << endl;
```

On most machines (considering the size of char), the above code displays this output:

```
sizeof(int): 4
sizeof(float): 4
sizeof(double): 8
sizeof(char): 1
sizeof(EmployeeRecord): 20
sizeof(i): 4
sizeof(f): 4
sizeof(d): 8
sizeof(c): 1
sizeof(er): 20
```

It is also important to note that the sizes of various types of variables can change depending on what system you're on. Check the DATA TYPES PAGE[245] for more information.

Syntactically, `sizeof` appears like a function call when taking the size of a type, but may be used without parentheses when taking the size of a variable type (e.g. `sizeof(int)`). Parentheses can be left out if the argument is a variable or array (e.g. `sizeof x`, `sizeof myArray`). Style guidelines vary on whether using the latitude to omit parentheses in the latter case is desirable.

Consider the next example:

245 Chapter 3.3.4 on page 139

```
#include <cstdio>

short func( short x )
{
  printf( "%d", x );
  return x;
}

int main()
{
  printf( "%d", sizeof( sizeof( func(256) ) ) );
}
```

Since `sizeof` does not evaluate anything at run time, the `func()` function is never called. All information needed is the return type of the function, the first `sizeof` will return the size of a short (the return type of the function) as the value 2 (in size_t, an integral type defined in the include file STDDEF.H) and the second `sizeof` will return 4 (the size of size_t returned by the first `sizeof`).

`sizeof` measures the size of an object in the simple sense of a contiguous area of storage; for types which include pointers to other storage, the indirect storage is *not* included in the value returned by `sizeof`. A common mistake made by programming newcomers working with C++ is to try to use `sizeof` to determine the length of a string; the `std::strlen` or `std::string::length` functions are more appropriate for that task.

`sizeof` has also found new life in recent years in template meta programming, where the fact that it can turn types into numbers, albeit in a primitive manner, is often useful, given that the TEMPLATE METAPROGRAMMING[246] environment typically does most of its calculations with types.

3.4.12 Dynamic memory allocation

Dynamic memory allocation is the allocation of MEMORY[247] storage for use in a COMPUTER PROGRAM[248] during the RUNTIME[249] of that program. It is a way of distributing ownership of limited memory resources among many pieces of data and code. Importantly, the amount of memory allocated is determined by the program at the time of allocation and need not be known in advance. A dynamic allocation exists until it is explicitly released, either by the programmer or by a

246 HTTP://EN.WIKIPEDIA.ORG/WIKI/TEMPLATE%20METAPROGRAMMING
247 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPUTER%20STORAGE
248 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPUTER%20PROGRAM
249 HTTP://EN.WIKIPEDIA.ORG/WIKI/RUNTIME

GARBAGE COLLECTOR[250] implementation; this is notably different from AUTOMATIC[251] and STATIC MEMORY ALLOCATION[252], which require advance knowledge of the required amount of memory and have a fixed duration. It is said that an object so allocated has *dynamic lifetime*.

The task of fulfilling an allocation request, which involves finding a block of unused memory of sufficient size, is complicated by the need to avoid both internal and external FRAGMENTATION[253] while keeping both allocation and deallocation EFFICIENT[254]. Also, the allocator's METADATA[255] can inflate the size of (individually) small allocations; CHUNKING[256] attempts to reduce this effect.

Usually, memory is allocated from a large pool of unused memory area called **the heap** (also called the **free store**). Since the precise location of the allocation is not known in advance, the memory is accessed indirectly, usually via a REFERENCE[257]. The precise algorithm used to organize the memory area and allocate and deallocate chunks is hidden behind an abstract interface and may use any of the methods described below.

You have probably wondered how programmers allocate memory efficiently without knowing, prior to running the program, how much memory will be necessary. Here is when the fun starts with dynamic memory allocation.

new and delete

For dynamic memory allocation we use the **new** and **delete** keywords, the old malloc from C functions can now be avoided but are still accessible for compatibility and low level control reasons.

As covered before, we assign values to pointers using the "address of" operator because it returns the address in memory of the variable or constant in the form of a pointer. Now, the "address of" operator is NOT the only operator that you can use to assign a pointer. You have yet another operator that returns a pointer, which is the new operator. The new operator allows the programmer to allocate memory

250 HTTP://EN.WIKIPEDIA.ORG/WIKI/GARBAGE%20COLLECTION%20%28COMPUTER%20SCIENCE%29
251 HTTP://EN.WIKIPEDIA.ORG/WIKI/AUTOMATIC%20MEMORY%20ALLOCATION
252 HTTP://EN.WIKIPEDIA.ORG/WIKI/STATIC%20MEMORY%20ALLOCATION
253 HTTP://EN.WIKIPEDIA.ORG/WIKI/FRAGMENTATION%20%28COMPUTER%29
254 HTTP://EN.WIKIPEDIA.ORG/WIKI/ALGORITHMIC_EFFICIENCY
255 HTTP://EN.WIKIPEDIA.ORG/WIKI/METADATA%20%28COMPUTING%29
256 HTTP://EN.WIKIPEDIA.ORG/WIKI/CHUNKING%20%28COMPUTING%29
257 HTTP://EN.WIKIPEDIA.ORG/WIKI/REFERENCE%20%28COMPUTER%20SCIENCE%29

for a specific data type, struct, class, etc., and gives the programmer the address of that allocated sect of memory in the form of a pointer. The new operator is used as an rvalue, similar to the "address of" operator. Take a look at the code below to see how the new operator works.

By assigning the pointers to an allocated sector of memory, rather than having to use a variable declaration, you basically override the "middleman" (the variable declaration). Now, you can allocate memory dynamically without having to know the number of variables you should declare.

```
int n = 10;
SOMETYPE *parray, *pS;
int *pint;

parray = new SOMETYPE[n];
pS = new SOMETYPE;
pint = new int;
```

If you looked at the above piece of code, you can use the new operator to allocate memory for arrays too, which comes quite in handy when we need to manipulate the sizes of large arrays and or classes efficiently. The memory that your pointer points to because of the new operator can also be "deallocated," not destroyed but rather, freed up from your pointer. The delete operator is used in front of a pointer and frees up the address in memory to which the pointer is pointing.

```
delete [] parray;// note the use of [] when destroying an array allocated with
 new
delete pint;
```

The memory pointed to by `parray` and `pint` have been freed up, which is a very good thing because when you're manipulating multiple large arrays, you try to avoid losing the memory someplace by leaking it. Any allocation of memory needs to be properly deallocated or a leak will occur and your program won't run efficiently. Essentially, every time you use the new operator on something, you should use the delete operator to free that memory before exiting. The delete operator, however, not only can be used to delete a pointer allocated with the new operator, but can also be used to "delete" a null pointer, which prevents attempts to delete non-allocated memory (this action compiles and does nothing).

You must keep in mind that `new T` and `new T()` are not equivalent. This will be more understandable after you are introduced to more complex types like classes, but keep in mind that when using `new T()` it will initialize the `T` memory location ("zero out") before calling the constructor (if you have non-initialized members variables, they will be initialized by default).

The **new** and **delete** operators do not have to be used in conjunction with each other within the same function or block of code. It is proper and often advised to write functions that allocate memory and other functions that deallocate memory. Indeed, the currently favored style is to release resources in object's destructors, using the so-called RESOURCE ACQUISITION IS INITIALIZATION[258] (RAII) idiom.

As we will see when we get to the Classes, a **class** destructor is the ideal location for its deallocator, it is often advisable to leave memory allocators out of classes' constructors. Specifically, using **new** to create an array of objects, each of which also uses **new** to allocate memory during its construction, often results in run-time errors. If a **class** or structure contains members which must be pointed at dynamically-created objects, it is best to sequentially initialize arrays of the parent object, rather than leaving the task to their constructors.

Note:
If possible you should use `new` and `delete` instead of `malloc` and `free`.

```
// Example of a dynamic array

const int b = 5;
int *a = new int[b];

//to delete
delete[] a;
```

The ideal way is to not use arrays at all, but rather the STL's vector type (a container similar to an array). To achieve the above functionality, you should do:

```
const int b = 5;
std::vector<int> a;
a.resize(b);

//to delete
a.clear();
```

Vectors allow for easy insertions even when "full." If, for example, you filled up a, you could easily make room for a 6th element like so:

```
int new_number = 99;
a.push_back( new_number );//expands the vector to fit the 6th element
```

258 HTTP://EN.WIKIPEDIA.ORG/WIKI/RAII

You can similarly dynamically allocate a rectangular multidimensional array (be careful about the type syntax for the pointers):

```
const int d = 5;
int (*two_d_array)[4] = new int[d][4];

//to delete
delete[] two_d_array;
```

You can also emulate a ragged multidimensional array (sub-arrays not the same size) by allocating an array of pointers, and then allocating an array for each of the pointers. This involves a loop.

```
const int d1 = 5, d2 = 4;
int **two_d_array = new int*[d1];
for( int i = 0; i < d1; ++i)
  two_d_array[i] = new int[d2];

//to delete
for( int i = 0; i < d1; ++i)
  delete[] two_d_array[i];

delete[] two_d_array;
```

3.4.13 Logical operators

The operators **and** (can also be written as **&&**) and **or** (can also be written as ||) allow two or more conditions to be chained together. The **and** operator checks whether all conditions are true and the **or** operator checks whether at least one of the conditions is true. Both operators can also be mixed together in which case the order in which they appear from left to right, determines how the checks are performed. Older versions of the C++ standard used the keywords **&&** and || in place of **and** and **or**. Both operators are said to *short circuit*. If a previous **and** condition is false, later conditions are not checked. If a previous **or** condition is true later conditions are not checked.

Note:
The iso646.h header file is part of the C standard library, since 1995, as an amendment to the C90 standard. It defines a number of macros which allow programmers to use C language bitwise and logical operators in textual form, which, without the header file, cannot be quickly or easily typed on some international and non-QWERTY keyboards. These symbols are keywords in the ISO C++ programming language and do not require the inclusion of a header file. For consistency, however, the C++98 standard provides the header <ciso646>. On MS Visual Studio that historically implements nonstandard language extensions this is the only way to enable these keywords (via macros) without disabling the extensions.

The **not** (can also be written as **!**) operator is used to return the inverse of one or more conditions.

- **Syntax**:

condition1 **and***condition2*
condition1 **or***condition2*
not *condition*

- **Examples**:

When something should not be true. It is often combined with other conditions. If x>5 but not x = 10, it would be written:

```
if ((x > 5) and not (x == 10)) // if (x greater than 5) and ( not (x equal to 10)
 )
{
  //...code...
}
```

When all conditions must be true. If x must be between 10 and 20:

```
if (x > 10 and x < 20) // if x greater than 10 and x less than 20
{
  //....code...
}
```

When at least one of the conditions must be true. If x must be equal to 5 or equal to 10 or less than 2:

```
if (x == 5 or x == 10 or x < 2) // if x equal to 5 or x equal to 10 or x less
 than 2
{
```

```
  //...code...
}
```

When at least one of a group of conditions must be true. If x must be between 10 and 20 or between 30 and 40.

```
if ((x >= 10 and x <= 20) or (x >= 30 and x <= 40)) // >= -> greater or equal
 etc...
{
  //...code...
}
```

Things get a bit more tricky with more conditions. The trick is to make sure the parenthesis are in the right places to establish the order of thinking intended. However, when things get this complex, it can often be easier to split up the logic into nested if statements, or put them into bool variables, but it is still useful to be able to do things in complex boolean logic.

Parenthesis around `x > 10` and around `x < 20` are implied, as the < operator has a higher precedence than **and**. First `x` is compared to 10. If `x` is greater than 10, `x` is compared to 20, and if `x` is also less than 20, the code is executed.

and (&&)

statement1	*statement2*	**and**
T	T	T
T	F	F
F	T	F
F	F	F

The logical AND operator, **and**, compares the left value and the right value. If both *statement1* and *statement2* are true, then the expression returns TRUE. Otherwise, it returns FALSE.

```
if ((var1 > var2) and (var2 > var3))
{
  std::cout << var1 " is bigger than " << var2 << " and " << var3 << std::endl;
}
```

In this snippet, the **if** statement checks to see if *var1* is greater than *var2*. Then, it checks if *var2* is greater than *var3*. If it is, it proceeds by telling us that *var1* is bigger than both *var2* and *var3*.

Note:
The logical AND operator **and** is sometimes written as **&&**, which is not the same as the address operator and the bitwise AND operator, both of which are represented with **&**

or (||)

statement1	*statement2*	**or**
T	T	T
T	F	T
F	T	T
F	F	F

The logical OR operator is represented with **or**. Like the logical AND operator, it compares *statement1* and *statement2*. If either *statement1* or *statement2* are true, then the expression is true. The expression is also true if both of the statements are true.

```
if ((var1 > var2) or (var1 > var3))
{
  std::cout << var1 " is either bigger than " << var2 << " or " << var3 <<
 std::endl;
}
```

Let's take a look at the previous expression with an OR operator. If *var1* is bigger than either *var2* or *var3* or both of them, the statements in the **if** expression are executed. Otherwise, the program proceeds with the rest of the code.

not (!)

The logical NOT operator, **not**, returns TRUE if the statement being compared is not true. Be careful when you're using the NOT operator, as well as any logical operator.

```
not x > 10
```

The logical expressions have a higher precedence than normal operators. Therefore, it compares whether "not x" is greater than 10. However, this statement always returns false, no matter what "x" is. That's because the logical expressions only return boolean values(1 and 0).

3.4.14 Conditional Operator

Conditional operators (also known as ternary operators) allow a programmer to check: if (x is more than 10 and eggs is less than 20 and x is not equal to a...).

Most operators compare two variables; the one to the left, and the one to the right. However, C++ also has a ternary operator (sometimes known as the conditional operator), **?:** which chooses from two expressions based on the value of a condition expression. The basic syntax is:

```
condition-expression ? expression-if-true : expression-if-false
```

If *condition-expression* is true, the expression returns the value of *expression-if-true*. Otherwise, it returns the value of *expression-if-false*. Because of this, the ternary operator can often be used in place of the **if** expression.

Note:
The use of the ternary operator versus the **if** expression often depends on the level of complexity and overall impact of the logical decision tree, using the **if** expression in convoluted or less than obvious situations should be preferred as it can not only be more clearly written but easier to understand, thus avoiding simple logical errors that would otherwise be hard to perceive.

- **For example:**

```
int foo = 8;
std::cout << "foo is " << (foo < 10 ? "smaller than" : "greater than or equal
to") << " 10." << std::endl;
```

The output will be "foo is smaller than 10.".

3.5 Type Conversion

Type conversion (often a result of *type casting*) refers to changing an entity of one DATA TYPE[259], expression, function argument, or return value into another. This is done to take advantage of certain features of type hierarchies. For instance, values from a more limited set, such as integers, can be stored in a more compact

259 Chapter 3.3.4 on page 139

format and later converted to a different format enabling operations not previously possible, such as division with several decimal places' worth of accuracy. In the OBJECT-ORIENTED[260] programming paradigm, type conversion allows programs also to treat objects of one type as one of another. One must do it carefully as type casting can lead to loss of data.

Note:
The Wikipedia article about STRONGLY TYPED[a] suggests that there is not enough consensus on the term "strongly typed" to use it safely. So you should re-check the intended meaning carefully, the above statement is what C++ programmers refer as *strongly typed* in the language scope.

a HTTP://EN.WIKIPEDIA.ORG/WIKI/STRONGLY-TYPED_PROGRAMMING_LANGUAGE

3.5.1 Automatic type conversion

Automatic type conversion (or standard conversion) happens whenever the compiler expects data of a particular type, but the data is given as a different type, leading to an automatic conversion by the compiler without an explicit indication by the programmer.

Note:
This is not "casting" or explicit type conversions. There is no such thing as an "automatic cast".

When an expression requires a given type that cannot be obtained through an implicit conversion or if more than one standard conversion creates an ambiguous situation, the programmer must explicitly specify the target type of the conversion. If the conversion is impossible it will result in an error or warning at compile time. Warnings may vary depending on the compiler used or compiler options.

This type of conversion is useful and relied upon to perform integral promotions, integral conversions, floating point conversions, floating-integral conversions, arithmetic conversions, pointer conversions.

```
int a = 5.6;
float b = 7;
```

260 HTTP://EN.WIKIPEDIA.ORG/WIKI/OBJECT-ORIENTED

In the example above, in the first case an expression of type float is given and automatically interpreted as an integer. In the second case (more subtle), an integer is given and automatically interpreted as a float.

There are two types of automatic type conversions between numeric types: promotion and conversion. Numeric promotion causes a simple type conversion whenever a value is used, while more complex numeric conversions can take place if the context of the expression requires it.

Any automatic type conversion is an implicit conversion if not done explicitly in the source code.

Automatic type conversions (implicit conversions) can also occur in the implicit "decay" from an array to a corresponding pointer type based or as a USER DEFINED BEHAVIOR[261]. We will cover that after we introduce classes (user defined types) as the automatic type conversions of references (derived class reference to base class reference) and pointer-to-member (from pointing to member of a base class to pointing to member of a derived class).

Promotion

A numeric promotion is the conversion of a value to a type with a wider range that happens whenever a value of a narrower type is used. Values of integral types narrower than `int` (`char`, `signed char`, `unsigned char`, `short int` and `unsigned short`) will be promoted to `int` if possible, or `unsigned int` if `int` can't represent all the values of the source type. Values of `bool` type will also be converted to `int`, and in particular `true` will get promoted to 1 and `false` to 0.

```
// promoting short to int
short left = 12;
short right = 23;

short total = left + right;
```

In the code above, the values of `left` and `right` are both of type `short` and could be added and assigned as such. However, in C++ they will each be promoted to `int` before being added, and the result converted back to `short` afterwards. The reason for this is that the `int` type is designed to be the most natural integer representation on the machine architecture, so requiring that the compiler do its calculations with smaller types may cause an unnecessary performance hit.

261 Chapter 4.3.1 on page 394

Since the C++ standard guarantees only the minimum sizes of the data types, the sizes of the types commonly vary between one architecture and another (and may even vary between one compiler and another). This is the reason why the compiler is allowed the flexibility to promote to `int` or `unsigned int` as necessary.

Promotion works in a similar way on floating-point values: a `float` value will be promoted to a `double` value, leaving the value unchanged.

Since promotion happens in cases where the expression does not require type conversion in order to be compiled, it can cause unexpected effects, for example in overload resolution:

```
void do_something(short arg)
{
    cout << "Doing something with a short" << endl;
}

void do_something(int arg)
{
    cout << "Doing something with an int" << endl;
}

int main(int argc, char **argv)
{
    short val = 12;

    do_something(val); // Prints "Doing something with a short"
    do_something(val * val); // Prints "Doing something with an int"
}
```

Since `val` is a `short`, you might expect that the expression `val * val` would also be a `short`, but in fact `val` is promoted to `int`, and the `int` overload is selected.

Numeric conversion

After any numeric promotion has been applied, the value can then be converted to another numeric type if required, subject to various constraints.

> **Note:**
> The standard guarantees that some conversions are possible without specifying what the exact result will be. This means that certain conversions that are legal can unexpectedly give different results using different compilers.

A value of any integer type can be converted to any other integer type, and a value of an enumeration type can be converted to an integer type. This only gets complicated when overflow is possible, as in the case where you convert from a larger

type to a smaller type. In the case of conversion to an unsigned type, overflow works in a nice predictable way: the result is the smallest unsigned integer congruent to the value being converted (modulo 2^n, where n is the number of bits in the destination type).

When converting to a signed integer type where overflow is possible, the result of the conversion depends on the compiler. Most modern compilers will generate a warning if a conversion occurs where overflow could happen. Should the loss of information be intended, the programmer may do explicit type casting to suppress the warning; bit masking may be a superior alternative.

Floating-point types can be converted between each other, but are even more prone to platform-dependence. If the value being converted can be represented exactly in the new type then the exact conversion will happen. Otherwise, if there are two values possible in the destination type and the source value lies between them, then one of the two values will be chosen. In all other cases the result is implementation-defined.

Floating-point types can be converted to integer types, with the fractional part being discarded.

```
double a = 12.5;
int b = a;

cout << b; // Prints "12"
```

Note:
If a floating-point value is converted to an integer and the result can't be expressed in the destination type, behavior is undefined by the C++ standard, meaning that your program may crash.

A value of an integer type can be converted to a floating point type. The result is exact if possible, otherwise it is the next lowest or next highest representable value (depending on the compiler).

3.5.2 Explicit type conversion (casting)

Explicit type conversion (casting) is the use of direct and specific notation in the source code to request a conversion or to specify a member from an overloaded class. There are cases where no automatic type conversion can occur or where the compiler is unsure about what type to convert to, those cases require explicit instructions from the programmer or will result in error.

Specific type casts

A set of casting operators have been introduced into the C++ language to address the shortcomings of the old C-style casts, maintained for compatibility purposes. Bringing with them a clearer syntax, improved semantics and type-safe conversions.

All of the casting operators share a similar syntax and as we will see are used in a manner similar to TEMPLATES[262], with these new keywords casting becomes easier to understand, find, and maintain.

The basic form of type cast

The basic explicit form of typecasting is the static cast.

A static cast looks like this:

```
static_cast<target type>(expression)
```

The compiler will try its best to interpret the *expression* as if it would be of type *type*. This type of cast will not produce a warning, even if the type is demoted.

```
int a = static_cast<int>(7.5);
```

The cast can be used to suppress the warning as shown above. `static_cast` cannot do all conversions; for example, it cannot remove const qualifiers, and it cannot perform "cross-casts" within a class hierarchy. It can be used to perform most numeric conversions, including conversion from a integral value to an enumerated type.

`static_cast`

The **static_cast** keyword can be used for any normal conversion between types. Conversions that rely on static (compile-time) type information. This includes any casts between numeric types, casts of pointers and references up the hierarchy, conversions with unary constructor, conversions with conversion operator. For conversions between numeric types no runtime checks if data fits the new type is performed. Conversion with unary constructor would be performed even if it is declared as explicit.

Syntax

262 Chapter 5 on page 483

```
TYPE static_cast<TYPE> (object);
```

It can also cast pointers or references down and across the hierarchy as long as such conversion is available and unambiguous. For example, it can cast `void*` to the appropriate pointer type or vice-versa. No runtime checks are performed.

```
BaseClass* a = new DerivedClass();
static_cast<DerivedClass*>(a)->derivedClassMethod();
```

Common usage of type casting

Performing arithmetical operations with varying types of data type without an explicit cast means that the compiler has to perform an implicit cast to ensure that the values it uses in the calculation are of the same type. Usually, this means that the compiler will convert all of the values to the type of the value with the highest precision.

The following is an integer division and so a value of 2 is returned.

```
float a = 5 / 2;
```

To get the intended behavior, you would either need to cast one or both of the constants to a **float**.

```
float a = static_cast<float>(5) / static_cast<float>(2);
```

Or, you would have to define one or both of the constants as a float.

```
float a = 5f / 2f;
```

`const_cast`

The **const_cast** keyword can be used to remove the *const* or *volatile* property from an object. The target data type must be the same as the source type, except (of course) that the target type doesn't have to have the same *const* qualifier. The type TYPE must be a pointer or reference type.

Syntax

```
TYPE const_cast<TYPE> (object);
```

For example, the following code uses **const_cast** to remove the *const* qualifier from a object:

```
class Foo {
public:
  void func() {} // a non-const member function
};

void someFunction( const Foo& f )  {
  f.func();      // compile error: cannot call a non-const
                 // function on a const reference
  Foo &fRef = const_cast<Foo&>(f);
  fRef.func();   // okay
}
```

dynamic_cast

The **dynamic_cast** keyword is used to casts a datum from one pointer or reference a of *polymorphic type* to another, similar to `static_cast` but performing a type safety check at runtime to ensure the validity of the cast. Generally for the purpose of casting a pointer or reference up or down an inheritance chain (INHERITANCE HIERARCHY[263]) in a safe way, including performing so-called *cross casts*.

Syntax

```
TYPE& dynamic_cast<TYPE&> (object);
TYPE* dynamic_cast<TYPE*> (object);
```

The target type must be a pointer or reference type, and the expression must evaluate to a pointer or reference.

If you attempt to cast to a pointer type, and that type is not an actual type of the argument object, then the result of the cast will be `NULL`.

If you attempt to cast to a reference type, and that type is not an actual type of the argument object, then the cast will throw a `std::bad_cast` exception.

When it doesn't fail, dynamic cast returns a pointer or reference of the target type to the object to which expression referred.

```
struct A {
  virtual void f() { }
};
struct B : public A { };
```

263 Chapter 2.3.4 on page 20

```
struct C { };

void f () {
  A a;
  B b;

  A* ap = &b;
  B* b1 = dynamic_cast<B*> (&a);  // NULL, because 'a' is not a 'B'
  B* b2 = dynamic_cast<B*> (ap);  // 'b'
  C* c = dynamic_cast<C*> (ap);   // NULL.

  A& ar = dynamic_cast<A&> (*ap); // Ok.
  B& br = dynamic_cast<B&> (*ap); // Ok.
  C& cr = dynamic_cast<C&> (*ap); // std::bad_cast
}
```

`reinterpret_cast`

The **reinterpret_cast** keyword is used to simply cast one type bitwise to another. Any pointer or integral type can be casted to any other with reinterpret cast, easily allowing for misuse. For instance, with reinterpret cast one might, unsafely, cast an integer pointer to a string pointer. It should be used to cast between incompatible pointer types.

Syntax

```
TYPE reinterpret_cast<TYPE> (object);
```

The `reinterpret_cast<>()` is used for all non portable casting operations. This makes it simpler to find these non portable casts when porting an application from one OS to another.

The `reinterpret_cast<T>()` will change the type of an expression without altering its underlying bit pattern. This is useful to cast pointers of a particular type into a `void*` and subsequently back to the original type.

```
int a = 0xffe38024;
int* b = reinterpret_cast<int*>(a);
```

Old C-style casts

Other common type casts exist, they are of the form `type(expression)` (a functional, or function-style, cast) or `(type)expression` (often known simply as a C-style cast). The format of `(type)expression` is more common in C (where it is the only cast notation). It has the basic form:

```
int i = 10;
```

```
long l;

l = (long)i; //C programming style cast
l = long(i); //C programming style cast in functional form (preferred by some C++
 programmers)
            //note: initializes a new long to i, this is not an explicit cast as
 in the example above
            //however an implicit cast does occur. i = long((long)i);
```

A C-style cast can, in a single line of source code, make two conversions. For instance remove a variable consteness and alter its type. In C++, the old C-style casts are retained for backwards compatibility.

```
const char string[]="1234";
function( (unsigned char*) string ); //remove const, add unsigned
```

There are several shortcomings in the old C-style casts:

1. They allows casting practically any type to any other type. Leading to lots of unnecessary trouble, even to creating source code that will compile but not to the intended result.
2. The syntax is the same for every casting operation. Making it impossible for the compiler and users to tell the intended purpose of the cast.
3. Hard to identify in the source code.

The C++ specific cast keyword are more controlled. Some will make the code safer since they will enable to catch more errors at compile-time, and all are easier to search, identify and maintain in the source code. Performance wise they are the same with the exception of `dynamic_cast`, for which there is no C equivalent. This makes them generally preferred. [264]

3.6 Control flow statements

Usually a program is not a linear sequence of instructions. It may repeat code or take decisions for a given path-goal relation. Most programming languages have **control flow** statements (constructs) which provide some sort of control structures that serve to specify order to what has to be done to perform our program that allow variations in this sequential order:

- statements may only be obeyed under certain conditions (conditionals),
- statements may be obeyed repeatedly under certain conditions (loops),
- a group of remote statements may be obeyed (subroutines).

264 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

Logical Expressions as conditions

Logical expressions can use logical operators in loops and conditional statements as part of the conditions to be met.

3.6.1 Exceptional and unstructured control flow

Some instructions have no particular structure but will have an exceptional usefulness in shaping how other control flow statements are structured, a special care must be taken to prevent unstructured and confusing programming.

break

A **break** will force the exiting of the present loop iteration into the next statement outside of the loop. It has no usefulness outside of a loop structure except for the **switch** control statement.

continue

The **continue** instruction is used inside loops where it will stop the current loop iteration, initiating the next one.

`goto`

The `goto` keyword is discouraged as it makes it difficult to follow the program logic, this way inducing to errors. The `goto` statement causes the current thread of execution to jump to the specified label.

Syntax

```
label:
  statement(s);

goto label;
```

In some rare cases, the `goto` statement allows to write uncluttered code, for example, when handling multiple exit points leading to the cleanup code at a function

exit (and neither exception handling or object destructors are better options). Except in those rare cases, the use of unconditional jumps is a frequent symptom of a complicated design, as the presence of many levels of nested statements.

In exceptional cases, like heavy optimization, a programmer may need more control over code behavior; a `goto` allows the programmer to specify that execution flow jumps directly and unconditionally to a desired label. A *label* is the name given to a label statement elsewhere in the function.

Note:
There is a classic paper in software engineering by W. A. WULF[a] called "A CASE AGAINST THE GOTO"[b], presented in the 25th ACM[c] National Conference in October 1972, a time when the debate about `goto` statements was reaching its peak. In this paper Wulf defends that `goto` statements should be regarded as dangerous. Wulf is also known by one of his comments regarding efficiency: "More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason -- including blind stupidity.".

a HTTP://EN.WIKIPEDIA.ORG/WIKI/WILLIAM%20WULF
b HTTP://PORTAL.ACM.ORG/CITATION.CFM?ID=1241523
c HTTP://EN.WIKIPEDIA.ORG/WIKI/ASSOCIATION%20FOR%20COMPUTING%20MACHINERY

A `goto` can, for example, be used to break out of two nested loops. This example breaks after replacing the first encountered non-zero element with zero.

```
for (int i = 0; i < 30; ++i) {
  for (int j = 0; j < 30; ++j) {
    if (a[i][j] != 0) {
       a[i][j] = 0;
       goto done;
    }
  }
}
done:
/* rest of program */
```

Although simple, they quickly lead to illegible and unmaintainable code.

```
// snarled mess of gotos

int i = 0;
  goto test_it;
body:
  a[i++] = 0;
test_it:
  if (a[i])
```

```
    goto body;
/* rest of program */
```

is much less understandable than the equivalent:

```
for (int i = 0; a[i]; ++i) {
  a[i] = 0;
}
/* rest of program */
```

Gotos are typically used in functions where performance is critical or in the output of machine-generated code (like a parser generated by YACC[265].)

The `goto` statement should almost always be avoided, there are rare cases when it enhances the readability of code. One such case is an "error section".

Example

```
#include <new>
#include <iostream>

int *my_allocated_1;
char *my_allocated_2, *my_allocated_3;
my_allocated_1 = new (std::nothrow) int[500];

if (my_allocated_1 == NULL)
{
  std::cerr << "error in allocated_1" << std::endl;
  goto error;
}

my_allocated_2 = new (std::nothrow) char[1000];

if (my_allocated_2 == NULL)
{
  std::cerr << "error in allocated_2" << std::endl;
  goto error;
}

my_allocated_3 = new (std::nothrow) char[1000];

if (my_allocated_3 == NULL)
{
  std::cerr << "error in allocated_3" <<std::endl;
  goto error;
}
return 0;

error:
  if (my_allocated_1) delete [] my_allocated_1;
  if (my_allocated_2) delete [] my_allocated_2;
  if (my_allocated_3) delete [] my_allocated_3;
  return 1;
```

265 HTTP://EN.WIKIPEDIA.ORG/WIKI/YACC

This construct avoids hassling with the origin of the error and is cleaner than an equivalent construct with control structures. It is thus less error prone.

Note:
While the above example shows a reasonable use of **goto**s, it is uncommon in practice. *Exceptions* handle such cases in a clearer, more effective and more organized way. This will be discussed in "Exception Handling" in detail. Using RAII to manage resources such as memory also avoids the need for most of the explicit cleanup code that is shown above.

abort(), exit() and atexit()

As we will see later the STANDARD C LIBRARY[266] that is included in C++ also supplies some useful functions that can alter the flow control. Some will permit you to terminate the execution of a program, enabling you to set up a return value or initiate special tasks upon the termination request. You will have to jump ahead into the ABORT()[267] - EXIT()[268] - ATEXIT()[269] sections for more information.

3.6.2 Conditionals

There is likely no meaningful program written in which a computer does not demonstrate basic decision-making skills based upon certain set conditions. It can actually be argued that there is no meaningful human activity in which no decision-making, instinctual or otherwise, takes place. For example, when driving a car and approaching a traffic light, one does not think, "I will continue driving through the intersection." Rather, one thinks, "I will stop if the light is red, go if the light is green, and if yellow go only if I am traveling at a certain speed a certain distance from the intersection." These kinds of processes can be simulated using conditionals.

A conditional is a statement that instructs the computer to execute a certain block of code or alter certain data only if a specific condition has been met.

The most common conditional is the if-else statement, with conditional expressions and switch-case statements typically used as more shorthanded methods.

266 Chapter 3.7.10 on page 264
267 Chapter 3.7.11 on page 356
268 Chapter 3.7.11 on page 358
269 Chapter 3.7.11 on page 357

if (Fork branching)

The if-statement allows one possible path choice depending on the specified conditions.

Syntax

```
if (condition)
{
  statement;
}
```

Semantic

First, the condition is evaluated:

- if *condition* is true, *statement* is executed before continuing with the body.
- if *condition* is false, the program skips *statement* and continues with the rest of the program.

Note:
The condition in an **if** statement can be any code that resolves in any expression that will evaluate to either a boolean, or a null/non-null value; you can declare variables, nest statements, etc. This is true to other flow control conditionals (ie: while), but is generally regarded as bad style, since it only benefit is ease of typing by making the code less readable.
This characteristic can easily lead simple errors, like tipping a=b (assign a value) in place of a a==b (condition). This has resulted in the adoption of a coding practice that would automatically put the errors in evidence, by inverting the expression (or using constant variables) the compiler will generate an error.
Recent compilers support the detection of such events and generate compilation warnings.

Example

```
if(condition)
{
  int x; // Valid code
  for(x = 0; x < 10; ++x) // Also valid.
    {
      statement;
    }
}
```

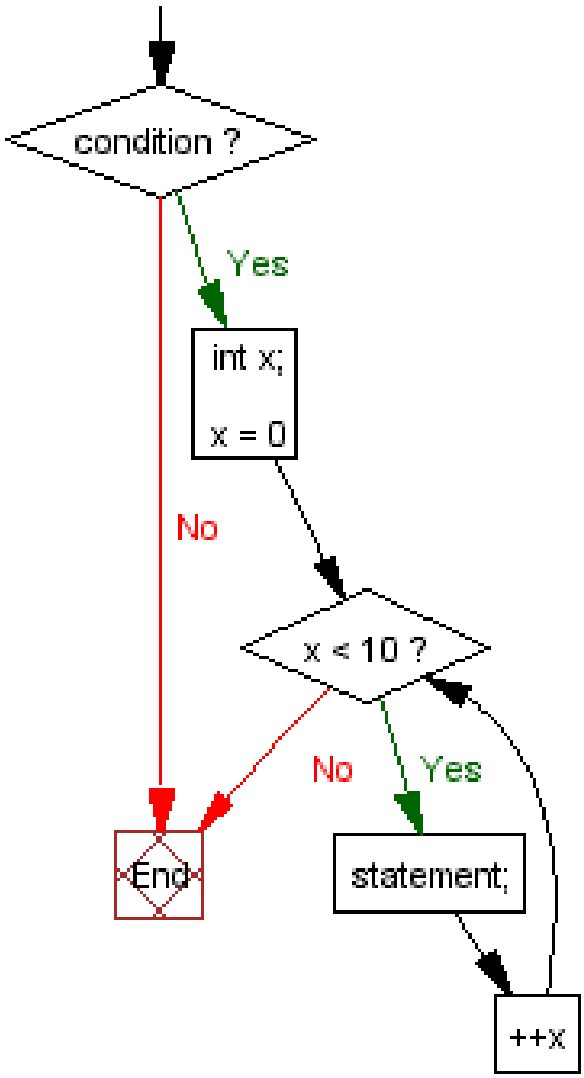


Figure 20: flowchart from the example

Note:
If you wish to avoid typing std::cout, std::cin, or std::endl; all the time, you may include **using namespace** std at the beginning of your program since cout, cin, and endl are members of the *std* namespace.

Sometimes the program needs to choose one of two possible paths depending on a condition. For this we can use the if-else statement.

```
if (user_age < 18)
{
    std::cout << "People under the age of 18 are not allowed." << std::endl;
}
else
{
    std::cout << "Welcome to Caesar's Casino!" << std::endl;
}
```

Here we display a message if the user is under 18. Otherwise, we let the user in. The if part is executed only if 'user_age' is less than 18. In other cases (when 'user_age' is greater than or equal to 18), the else part is executed.

if conditional statements may be chained together to make for more complex condition branching. In this example we expand the previous example by also checking if the user is above 64 and display another message if so.

```
if (user_age < 18)
{
  std::cout << "People under the age of 18 are not allowed." << std::endl;
}
else if (user_age > 64)
{
  std::cout << "Welcome to Caesar's Casino! Senior Citizens get 50% off." <<
 std::endl;
}
else
{
  std::cout << "Welcome to Caesar's Casino!" << std::endl;
}
```

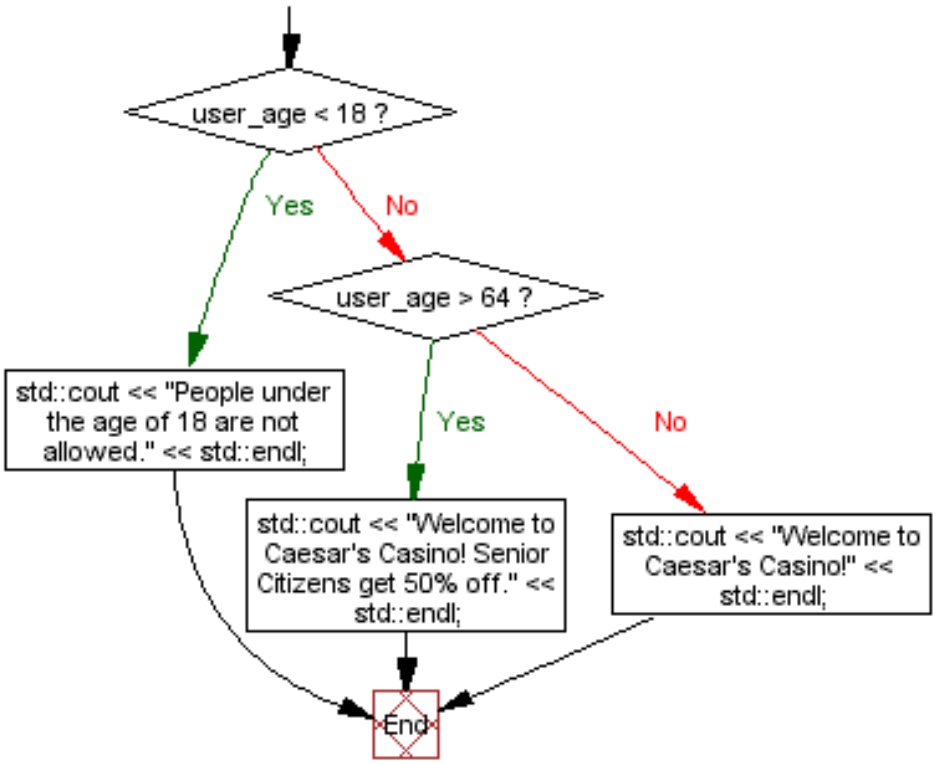


Figure 21: flowchart from the example

Note:

- **break** and **continue** do not have any relevance to an **if** or **else**.
- Although you can use multiple **else if** statements, when handling many related conditions it is recommended that you use the **switch** statement, which we will be discussing next.

switch (Multiple branching)

The switch statement branches based on specific integer values.

```
switch (integer expression) {
    case label1:
        statement(s)
        break;
    case label2:
        statement(s)
        break;
    /* ... */
    default:
        statement(s)
}
```

As you can see in the above scheme the case and default have a "break;" statement at the end of block. This expression will cause the program to exit from the switch, if break is not added the program will continue execute the code in other cases even when the integer expression is not equal to that case. This can be exploited in some cases as seen in the next example.

We want to separate an input from digit to other characters.

```
char ch = cin.get(); //get the character
switch (ch) {
    case '0':
        // do nothing fall into case 1
    case '1':
       // do nothing fall into case 2
    case '2':
      // do nothing fall into case 3
    /* ... */
    case '8':
      // do nothing fall into case 9
    case '9':
        std::cout << "Digit" << endl; //print into stream out
        break;
    default:
        std::cout << "Non digit" << endl; //print into stream out
}
```

In this small piece of code for each digit below ’9’ it will propagate through the cases until it will reach case ’9’ and print "digit".

If not it will go straight to the default case there it will print "Non digit"

Note:

- Be sure to use **break** commands unless you want multiple conditions to have the same action. Otherwise, it will "fall through" to the next set of commands.
- **break** can only break out of the innermost level. If for example you are inside a **switch** and need to break out of a enclosing `for` loop you might well consider adding a boolean as a flag, and check the flag after the **switch** block instead of the alternatives available. (Though even then, refactoring the code into a separate function and returning from that function might be cleaner depending on the situation, and with inline functions and/or smart compilers there need not be any runtime overhead from doing so.)
- **continue** is not relevant to **switch** block. Calling **continue** within a **switch** block will lead to the "continue" of the loop which wraps the **switch** block.

3.6.3 Loops (iterations)

A loop (also referred to as an iteration or repetition) is a sequence of statements which is specified once but which may be carried out several times in succession. The code "inside" the loop (the *body* of the loop) is obeyed a specified number of times, or once for each of a collection of items, or until some condition is met.

ITERATION[270] is the repetition of a process, typically within a computer program. Confusingly, it can be used both as a general term, synonymous with repetition, and to describe a specific form of repetition with a MUTABLE[271] state.

When used in the first sense, RECURSION[272] is an example of iteration.

However, when used in the second (more restricted) sense, iteration describes the style of programming used in imperative programming languages. This contrasts with recursion, which has a more declarative approach.

Due to the nature of C++ there may lead to an even bigger problems when differentiating the use of the word, so to simplify things use "**loops**" to refer to simple recursions as described in this section and use *iteration* or *iterator*[273] (the "one"

270 HTTP://EN.WIKIPEDIA.ORG/WIKI/ITERATION
271 HTTP://EN.WIKIPEDIA.ORG/WIKI/MUTABLE%20OBJECT
272 HTTP://EN.WIKIPEDIA.ORG/WIKI/RECURSION
273 HTTP://EN.WIKIPEDIA.ORG/WIKI/ITERATOR

that performs an *iteration*) to class *iterator* (or in relation to objects/classes) as used in the STL.

Infinite Loops

Sometimes it is desirable for a program to loop forever, or until an exceptional condition such as an error arises. For instance, an event-driven program may be intended to loop forever handling events as they occur, only stopping when the process is killed by the operator.

More often, an infinite loop is due to a programming error in a condition-controlled loop, wherein the loop condition is never changed within the loop.

```
// as we will see, these are infinite loops...
while (1) { }

// or

for (;;) { }
```

> **Note:**
> When the compiler optimizes the source code, all statement after the detected infinite loop (that will never run), will be ignored. A compiler warning is generally given on detecting such cases.

Condition-controlled loops

Most programming languages have constructions for repeating a loop until some condition changes.

Condition-controlled loops are divided into two categories Preconditional or Entry-Condition that place the test at the start of the loop, and Postconditional or Exit-Condition iteration that have the test at the end of the loop. In the former case the body may be skipped completely, while in the latter case the body is always executed at least once.

In the condition controlled loops, the keywords *break* and *continue* take significance. The break keyword causes an exit from the loop, proceeding with the rest of the program. The *continue* keyword terminates the current iteration of the loop, the loop proceeds to the next iteration.

while (Preconditional loop)

Syntax

```
while (''condition'') ''statement''; ''statement2'';
```

Semantic First, the condition is evaluated:

1. if *condition* is true, *statement* is executed and *condition* is evaluated again.
2. if *condition* is false continues with *statement2*

Remark: *statement* can be a block of code { ... } with several instructions.

What makes 'while' statements different from the 'if' is the fact that once the body (referred to as *statement* above) is executed, it will go back to 'while' and check the condition again. If it is true, it is executed again. In fact, it will execute as many times as it has to until the expression is false.

Example 1

```
#include <iostream>
using namespace std;

int main()
{
  int i=0;
  while (i<10) {
    cout << "The value of i is " << i << endl;
    i++;
  }
  cout << "The final value of i is : " << i << endl;
  return 0;
}
```

Execution

```
   The value of i is 0
   The value of i is 1
   The value of i is 2
   The value of i is 3
   The value of i is 4
   The value of i is 5
   The value of i is 6
   The value of i is 7
   The value of i is 8
   The value of i is 9
   The final value of i is 10
```

Example 2

```
// validation of an input
#include <iostream>
using namespace std;

int main()
{
  int a;
  bool ok=false;
  while (!ok) {
    cout << "Type an integer from 0 to 20 : ";
    cin >> a;
    ok = ((a>=0) && (a<=20));
    if (!ok) cout << "ERROR - ";
  }
  return 0;
}
```

Execution

```
Type an integer from 0 to 20 : 30
ERROR - Type an integer from 0 to 20 : 40
ERROR - Type an integer from 0 to 20 : -6
ERROR - Type an integer from 0 to 20 : 14
```

do-while (Postconditional loop)

Syntax

```
do {
  statement(s)
} while (condition);

statement2;
```

Semantic

1. *statement(s)* are executed.
2. *condition* is evaluated.
3. if *condition* is true goes to 1).
4. if *condition* is false continues with *statement2*

The do - while loop is similar in syntax and purpose to the while loop. The construct moves the test that continues condition of the loop to the end of the code block so that the code block is executed at least once before any evaluation.

Example

```
#include <iostream>
```

```
using namespace std;

int main()
{
  int i=0;

  do {
    cout << "The value of i is " << i << endl;
    i++;
  } while (i<10);

  cout << "The final value of i is : " << i << endl;
  return 0;
}
```

Execution

```
The value of i is 0
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
The value of i is 6
The value of i is 7
The value of i is 8
The value of i is 9
The final value of i is 10
```

`for` (Preconditional and counter-controlled loop)

The **for** keyword is used as special case of a pre-conditional loop that supports constructors for repeating a loop only a certain number of times in the form of a step-expression that can be tested and used to set a *step size* (the rate of change) by incrementing or decrementing it in each loop.

Syntax

```
for (initialization ; condition; step-expression)
  statement(s);
```

The for construct is a general looping mechanism consisting of 4 parts:

1. . the initialization, which consists of 0 or more comma-delimited variable initialization statements

2. . the test-condition, which is evaluated to determine if the execution of the for loop will continue
3. . the increment, which consists of 0 or more comma-delimited statements that increment variables
4. . and the statement-list, which consists of 0 or more statements that will be executed each time the loop is executed.

Note:
Variables declared and initialized in the loop initialization (or body) are only valid in the SCOPE[a] of the loop itself.

a Chapter 3.1.10 on page 79

The for loop is equivalent to next while loop:

```
initialization
while( condition )
{
  statement(s);
  step-expression;
}
```

Note:
Each step of the loop (initialization, condition, and step-expression) can have more than one command, separated by a **,** (comma operator). *initialization*,*condition*, and *step expression* are all optional arguments. In C++ the comma is very rarely used as an operator. It is mostly used as a separator (ie. `int x, y;`).

Example 1

```
// a unbounded loop structure
for (;;)
{
  statement(s);
  if( statement(s) )
    break;
}
```

Example 2

```
// calls doSomethingWith() for 0,1,2,..9
for (int i = 0; i != 10; ++i)
{
```

```
  doSomethingWith(i);
}
```

can be rewritten as:

```
// calls doSomethingWith() for 0,1,2,..9
int i = 0;
while(i != 10)
{
  doSomethingWith(i);
  ++i;
}
```

The for loop is a very general construct, which can run unbounded loops (**Example 1**) and does not need to follow the rigid iteration model enforced by similarly named constructs in a number of more formal languages. C++ (just as modern C) allows variables (**Example 2**) to be declared in the initialization part of the for loop, and it is often considered good form to use that ability to declare objects only when they can be initialized, and to do so in the smallest scope possible. Essentially, the for and while loops are equivalent. Most for statements can also be rewritten as while statements.

3.7 Functions

A **FUNCTION**[274], which can also be referred to as SUBROUTINE[275], **procedure**, **subprogram** or even METHOD[276], carries out tasks defined by a sequence of statements called a STATEMENT BLOCK[277] that need only be written once and *called* by a program as many times as needed to carry out the same task.

Functions may depend on variables passed to them, called ARGUMENTS[278], and may pass results of a task on to the caller of the function, this is called the RETURN VALUE[279].

It is important to note that a **function** that exists in the GLOBAL SCOPE[280] can also be called **global function** and a function that is defined inside a class is called a **member function**. (The term **method** is commonly used in other programming languages to refer to things *like* member functions, but this can lead to confusion in

274 HTTP://EN.WIKIPEDIA.ORG/WIKI/SUBROUTINE
275 HTTP://EN.WIKIPEDIA.ORG/WIKI/SUBROUTINE
276 HTTP://EN.WIKIPEDIA.ORG/WIKI/METHOD_%28COMPUTER_SCIENCE%29
277 Chapter 3.1.6 on page 56
278 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23PARAMETERS%20AND%20ARGUMENTS
279 HTTP://EN.WIKIPEDIA.ORG/WIKI/RETURN%20STATEMENT
280 Chapter 3.1.9 on page 78

dealing with C++ which supports both virtual and non-virtual dispatch of member functions.)

Note:
When talking or reading about programming, you must consider the language background and the topic of the source. It is very rare to see a C++ programmer use the words **procedure** or **subprogram**, this will vary from language to language. In many programming languages the word **function** is reserved for subroutines that return a value, this is not the case with C++.

3.7.1 Declarations

A function must be declared before being used, with a name to identify it, what type of value the function returns and the types of any arguments that are to be passed to it. Parameters must be named and declare what type of value it takes. Parameters should always be passed as *const* if their arguments are not modified. Usually functions performs actions, so the name should make clear what it does. By using verbs in function names and following other naming conventions programs can be read more naturally.

The next example we define a function named `main` that returns an integer value `int` and takes no parameters. The content of the function is called the *body* of the function. The word `int` is a *keyword*. C++ keywords are *reserved words*, i.e., cannot be used for any purpose other than what they are meant for. On the other hand *main* is not a keyword and you can use it in many places where a keyword cannot be used (though that is not recommended, as confusion could result).

```
int main()
{
  // code
  return 0;
}
```

`inline`

The **inline** keyword declares an inline function, the declaration is a (non-binding) request to the compiler that a particular function be subjected to IN-LINE EXPANSION[281]; that is, it suggests that the compiler insert the complete body of the function in every context where that function is used and so it is used to avoid the overhead implied by making a CPU jump from one place in code to another and

281 HTTP://EN.WIKIPEDIA.ORG/WIKI/INLINE%20EXPANSION

back again to execute a subroutine, as is done in naive implementations of subroutines.

```
inline swap( int& a, int& b) { int const tmp(b); b=a; a=tmp; }
```

When a function definition is included in a class/struct definition, it will be an *implicit inline*, the compiler will try to automatically inline that function. No `inline` keyword is necessary in this case; it is legal, but redundant, to add the `inline` keyword in that context, and GOOD STYLE[282] is to omit it.

Example:

```
struct length
{
  explicit length(int metres) : m_metres(metres) {}
  operator int&() { return m_metres; }
  private:
  int m_metres;
};
```

Inlining can be an optimization, or a pessimization. It can increase code size (by duplicating the code for a function at multiple call sites) or can decrease it (if the code for the function, after optimization, is less than the size of the code needed to call a non-inlined function). It can increase speed (by allowing for more optimization and by avoiding jumps) or can decrease speed (by increasing code size and hence cache misses).

One important side-effect of inlining is that more code is then accessible to the optimizer.

Marking a function as inline also has an effect on linking: multiple definitions of an inline function are permitted (so long as each is in a different translation unit) so long as they are identical. This allows inline function definitions to appear in header files; defining non-inlined functions in header files is almost always an error (though function templates can also be defined in header files, and often are).

Mainstream C++ compilers like MICROSOFT VISUAL C++[283] and GCC[284] support an option that lets the compilers *automatically inline* any suitable function, even those that are not marked as inline functions. A compiler is often in a better position than a human to decide whether a particular function should be inlined; in particular, the compiler may not be willing or able to inline many functions that the human asks it to.

282 Chapter 3.1.7 on page 59
283 HTTP://EN.WIKIPEDIA.ORG/WIKI/VISUAL%20C%20PLUS%20PLUS
284 HTTP://EN.WIKIPEDIA.ORG/WIKI/GNU%20COMPILER%20COLLECTION

Excessive use of inlined functions can greatly increase coupling/dependencies and compilation time, as well as making header files less useful as documentation of interfaces.

Normally when calling a function, a program will evaluate and store the arguments, and then call (or branch to) the function's code, and then the function will later return back to the caller. While function calls are fast (typically taking much less than a microsecond on modern processors), the overhead can sometimes be significant, particularly if the function is simple and is called many times.

One approach which can be a performance optimization in some situations is to use so-called `inline` functions. Marking a function as `inline` is a request (sometimes called a hint) to the compiler to consider replacing a *call* to the function by a copy of the code of that function.

The result is in some ways similar to the use of the #define macro, but as MENTIONED BEFORE[285], macros can lead to problems since they are not evaluated by the PREPROCESSOR[286]. `inline` functions do not suffer from the same problems.

If the inlined function is large, this replacement process (known for obvious reasons as "inlining") can lead to "code bloat", leading to bigger (and hence usually slower) code. However, for small functions it can even reduce code size, particularly once a compiler's optimizer runs.

Note that the inlining process requires that the function's definition (including the code) must be available to the compiler. In particular, inline headers that are used from more than one source file must be completely defined within a header file (whereas with regular functions that would be an error).

The most common way to designate that a function is inline is by the use of the `inline` keyword. One must keep in mind that compilers can be configured to ignore this keyword and use their own optimizations.

Further considerations are given when dealing with INLINE MEMBER FUNCTION[287], this will be covered on the **OBJECT-ORIENTED PROGRAMMING CHAPTER**[288] .

285 Chapter 3.2.3 on page 98
286 Chapter 3.2.2 on page 98
287 Chapter 4.3.5 on page 409
288 Chapter 3.9 on page 384

3.7.2 Parameters and arguments

The function declaration defines its parameters. A parameter is a variable which takes on the meaning of a corresponding argument passed in a call to a function.

An argument represents the value you supply to a function parameter when you call it. The calling code supplies the arguments when it calls the function.

The part of the function declaration that declares the expected parameters is called the **parameter list** and the part of function call that specifies the arguments is called the **argument list**.

```
//Global functions declaration
int subtraction_function( int parameter1, int parameter2 ) { return ( parameter1
 - parameter2 ); }

//Call to the above function using 2 extra variables so the relation becomes more
 evident
int argument1 = 4;
int argument2 = 3;
int result = subtraction_function( argument1, argument2 );
// will have the same result as
int result = subtraction_function( 4, 3 );
```

Many programmers use parameter and argument interchangeably, depending on context to distinguish the meaning. In practice, distinguishing between the two terms is usually unnecessary in order to use them correctly or communicate their use to other programmers. Alternatively, the equivalent terms formal parameter and actual parameter may be used instead of parameter and argument.

3.7.3 Parameters

You can define a function with no parameters, one parameter, or more than one, but to use a call to that function with arguments you must take into consideration what is defined.

Empty parameter list

```
//Global functions with no parameters
void function() { /*...*/ }
//empty parameter declaration equivalent the use of void
void function( void ) ( /*...*/ }
```

Note:
This is the only valid case were *void* can be used as a parameter type, you can only derived types from *void* (ie: *void**).

Multiple parameters

The syntax for declaring and invoking functions with multiple parameters can be a source of errors. When you write the function definition, you must declare the type of each and every parameter.

```
// Example - function using two int parameters by value
void printTime (int hour, int minute) {
  std::cout << hour;
  std::cout << ":";
  std::cout << minute;
}
```

It might be tempting to write (int hour, minute), but that format is only legal for variable declarations, not for parameter declarations.

However, you do not have to declare the types of arguments when you call a function. (Indeed, it is an error to attempt to do so).

Example

```
int main void(){
int hour = 11;
    int minute = 59;
    printTime( int hour, int minute ); // WRONG!
    printTime( hour, minute ); // Right!
}
```

In this case, the compiler can tell the type of hour and minute by looking at their declarations. It is unnecessary and illegal to include the type when you pass them as arguments..

by pointer

A function may use pass by pointer when the object pointed to might not exist, that is, when you are giving either the address of a real object or NULL. Passing a pointer is not different to passing anything else. Its a parameter the same as any other. The characteristics of the pointer type is what makes it a worth distinguishing.

The passing a pointer to a function is very similar to passing it as a reference. It is used to avoid the overhead of copying, and the slicing problem (since child classes have a bigger memory footprint that the parent) that can occur when passing base class objects by value. This is also the preferred method in C (for historical reasons), were passing by pointer signifies that wanted to modify the original variable. In C++ it is preferred to use references to pointers and guarantee that the function before dereferencing it, verifies the pointer for validity.

```
#include <iostream>

void MyFunc( int *x )
{
  std::cout << *x << std::endl; // See next section for explanation
}

int main()
{
  int i;
  MyFunc( &i );

  return 0;
}
```

Since a reference is just an alias, it has exactly the same address as what it refers to, as in the following example:

```
#include <iostream>

void ComparePointers (int * a, int * b)
{
  if (a == b)
    std::cout<<"Pointers are the same!"<<std::endl;
  else
    std::cout<<"Pointers are different!"<<std::endl;
}

int main()
{
  int i, j;
  int& r = i;

  ComparePointers(&i, &i);
  ComparePointers(&i, &j);
  ComparePointers(&i, &r);
  ComparePointers(&j, &r);

  return 0;
}
```

This schizophrenic program will tell you that the pointers are the same, then that they are different, then the same, then different again.

Arrays are similar to pointers, remember?

Now might be a good time to reread the section on arrays. If you do not feel like flipping back that far, though, here's a brief recap: Arrays are blocks of memory space.

```
int my_array[5];
```

In the statement above, `my_array` is an area in memory big enough to hold five ints. To use an element of the array, it must be *dereferenced*. The third element in the array (remember they're zero-indexed) is `my_array[2]`. When you write `my_array[2]`, you're actually saying "give me the third integer in the array `my_array`". Therefore, `my_array` is an array, but `my_array[2]` is an `int`.

Passing a single array element

So let's say you want to pass one of the integers in your array into a function. How do you do it? Simply pass in the dereferenced element, and you'll be fine.

Example

```
#include <iostream>

void printInt(int printable){
  std::cout << "The int you passed in has value " << printable << std::endl;
}
int main(){
  int my_array[5];

  // Reminder: always initialize your array values!
  for(int i = 0; i < 5; i++)
    my_array[i] = i * 2;

  for(int i = 0; i < 5; i++)
    printInt(my_array[i]); // <-- We pass in a dereferenced array element
}
```

This program outputs the following:

```
  The int you passed in has value 0
  The int you passed in has value 2
  The int you passed in has value 4
  The int you passed in has value 6
  The int you passed in has value 8
```

This passes array elements just like normal integers, because array elements like `my_array[2]` are integers.

Passing a whole array

Well, we can pass single array elements into a function. But what if we want to pass a whole array? We can not do that directly, but you can treat the array as a pointer.

Example

```
#include <iostream>

void printIntArr(int *array_arg, int array_len){
  std::cout << "The length of the array is " << array_len << std::endl;
  for(int i = 0; i < array_len; i++)
    std::cout << "Array[" << i << "] = " << array_arg[i] << std::endl;
}

int main(){
  int my_array[5];

  // Reminder: always initialize your array values!
  for(int i = 0; i < 5; i++)
    my_array[i] = i * 2;

  printIntArr(my_array, 5);
}
```

Note:
Due to array-pointer interchangeability *in the context of parameter declarations only*, we can also declare pointers as arrays in function parameter lists. It is treated identically. For example, the first line of the function above can also be written as
`void printIntArr(int array_arg[], int array_len)`
It is important to note that even if it is written as `int array_arg[]`, the parameter is still a pointer of type **int ***. It is not an array; an array passed to the function will still be automatically converted to a pointer to its first element.

This will output the following:

```
The length of the array is 5
Array[0] = 0
Array[1] = 2
Array[2] = 4
```

```
  Array[3] = 6
  Array[4] = 8
```

As you can see, the array in main is accessed by a pointer. Now here's some important points to realize:

- Once you pass an array to a function, it is converted to a pointer so that function has no idea how to guess the length of the array. Unless you always use arrays that are the same size, you should always pass in the array length along with the array.
- You've passed in a POINTER. `my_array` is an array, not a pointer. If you change `array_arg` within the function, `my_array` does not change (i.e., if you set `array_arg` to point to a new array). But if you change any element of `array_arg`, you're changing the memory space pointed to by `array_arg`, which is the array `my_array`.

by reference

The same concept of references is used when passing variables.

Example

```
void foo( int &i )
{
  ++i;
}

int main()
{
  int bar = 5;   // bar == 5
  foo( bar );    // bar == 6
  foo( bar );    // bar == 7

  return 0;
}
```

Here we display one of the two common uses of references in function arguments -- they allow us to use the conventional syntax of passing an argument by value but manipulate the value in the caller.

Note:
If the parameter is a non-const reference, the caller expects it to be modified. If the function does not want to modify the parameter, a const reference should be used instead.

However there is a more common use of references in function arguments -- they can also be used to pass a handle to a large data structure without making multiple copies of it in the process. Consider the following:

```
void foo( const std::string & s ) // const reference, explained below
{
  std::cout << s << std::endl;
}

void bar( std::string s )
{
  std::cout << s << std::endl;
}

int main()
{
  std::string const text = "This is a test.";

  foo( text ); // doesn't make a copy of "text"
  bar( text ); // makes a copy of "text"

  return 0;
}
```

In this simple example we're able to see the differences in pass by value and pass by reference. In this case pass by value just expends a few additional bytes, but imagine for instance if `text` contained the text of an entire book.

The reason why we use a constant reference instead of a reference is the user of this function can assure that the value of the variable passed does not change within the function. We technically call this "const-to-reference".

The ability to pass it by reference keeps us from needing to make a copy of the string and avoids the ugliness of using a pointer.

> **Note:**
> It should also be noted that "const-to-reference" only makes sense for complex types -- classes and structs. In the case of ordinal types -- i.e. **int**, **float**, **bool**, etc. -- there is no savings in using a reference instead of simply using pass by value, and indeed the extra costs associated with indirection may make code using a reference slower than code that copies small objects.

Passing an array of fixed-length by using reference

In some case, a function requires an array of a specific length to work:

```
void func(int(&para)[4]);
```

Unlike the case of array changed into pointer above, the parameter is not a PLAIN array that can be changed into a pointer, but rather a reference to array with 4 `int`s. Therefore, only array of 4 `int`s, not array of any other length, not pointer to int, can be passed into this function. This helps you prevent buffer overflow errors because the array object is ALWAYS allocated unless you circumvent the type system by casting.

It can be used to pass an array without specifying the number of elements manually:

```
template<int n>void func(int(&para)[n]);
```

The compiler generates the value of length at compile time, inside the function, n stores the number of elements. However, the use of template generates code bloat.

In C++, a multi-dimensional array cannot be converted to a multi-level pointer, therefore, the code below is invalid:

```
// WRONG
void foo(int**matrix,int n,int m);
int main(){
        int data[10][5];
        // do something on data
        foo(data,10,5);
}
```

Although an int[10][5] can be converted to an (*int)[5], it cannot be converted to int**. Therefore you may need to hard-code the array bound in the function declaration:

```
// BAD
void foo(int(*matrix)[5],int n,int m);
int main(){
        int data[10][5];
        // do something on data
        foo(data,10,5);
}
```

To make the function more generic, templates and function overloading should be used:

```
// GOOD
template<int junk,int rubbish>void foo(int(&matrix)[junk][rubbish],int n,int m);
void foo(int**matrix,int n,int m);
int main(){
        int data[10][5];
        // do something on data
        foo(data,10,5);
}
```

The reason for having n and m in the first version is mainly for consistency, and also deal with the case that the array allocated is not used completely. It may also be used for checking buffer overflows by comparing n/m with junk/rubbish.

by value

When we want to write a function which the value of the argument is independent to the passed variable, we use pass-by-value approach.

```
int add(int num1, int num2)
{
 num1 += num2; // change of value of "num1"
 return num1;
}

int main()
{
 int a = 10, b = 20, ans;
 ans = add(a, b);
 std::cout << a << " + " << b << " = " << ans << std::endl;
 return 0;
}
```

Output:

```
  10 + 20 = 30
```

The above example shows a property of pass-by-value, the arguments are copies of the passed variable and only in the SCOPE[289] of the corresponding function. This means that we have to afford the cost of copying. However, this cost is usually considered only for larger and more complex variables.

In this case, the values of "a" and "b" are copied to "num1" and "num2" on the function "add()". We can see that the value of "num1" is changed in line 3. However, we can also observe that the value of "a" is kept after passed to this function.

Constant Parameters

The keyword `const` can also be used as a guarantee that a function will not modify a value that is passed in. This is really only useful for references and pointers (and

289 Chapter 3.1.9 on page 78

not things passed by value), though there's nothing syntactically to prevent the use of `const` for arguments passed by value.

Take for example the following functions:

```
void foo( const std::string &s )
{
   s.append("blah"); // ERROR -- we can't modify the string

   std::cout << s.length() << std::endl; // fine
}

void bar( const Widget *w )
{
    w->rotate(); // ERROR - rotate wouldn't be const

    std::cout << w->name() << std::endl; // fine
}
```

In the first example we tried to call a non-const method -- `append()` -- on an argument passed as a `const` reference, thus breaking our agreement with the caller not to modify it and the compiler will give us an error.

The same is true with `rotate()`, but with a `const` pointer in the second example.

Default values

Parameters in C++ functions (including member functions and constructors) can be declared with default values, like this

```
int foo (int a, int b = 5, int c = 3);
```

Then if the function is called with fewer arguments (but enough to specify the arguments without default values), the compiler will assume the default values for the missing arguments at the end. For example, if I call

```
foo(6, 1)
```

that will be equivalent to calling

```
foo(6, 1, 3)
```

In many situations, this saves you from having to define two separate functions that take different numbers of parameters, which are almost identical except for a default value.

The "value" that is given as the default value is often a constant, but may be any valid expression, including a function call that performs arbitrary computation.

Default values can only be given for the last arguments; i.e. you cannot give a default value for a parameter that is followed by a parameter that does not have a default value, since it will never be used.

Once you define the default value for a parameter in a function declaration, you cannot re-define a default value for the same parameter in a later declaration, even if it is the same value.

Ellipsis (...) as a parameter

If the parameter list ends with an ellipsis, it means that the arguments number must be equal or greater than the number of parameters specified. It will in fact create a variadic function, a function of variable arity; that is, one which can take different numbers of arguments.

Note:
The variadic function feature is going to be readdressed in the upcoming C++ language standard, C++0x; with the possible inclusion of variatic macros and the ability to create variadic template classes and variadic template functions. Variadic templates will finally allow the creation of true tuple classes in C++.

3.7.4 Returning values

When declaring a function, you must declare it in terms of the type that it will return, this is done in three steps, in the function declaration, the function implementation (if distinct) and on the body of the same function with the `return` keyword.

Functions with results

You might have noticed by now that some of the functions yield results. Other functions perform an action but don't `return` a value.

Other ways to get a value from a function is to use a pointer or a reference as argument or use a global variable

Get more that a single value from a function

The return type determines the capacity, any type will work from an array or a std::vector, a struct or a class, it is only restricted by the return type you chose.

That raises some questions

- What happens if you call a function and you don't do anything with the result (i.e. you don't assign it to a variable or use it as part of a larger expression)?
- What happens if you use a function without a result as part of an expression, like newLine() + 7?
- Can we write functions that yield results, or are we stuck with things like newLine and printTwice?

The answer to the third question is "yes, you can write functions that returns values,". For now I will leave it up to you to answer the other two questions by trying them out. Any time you have a question about what is legal or illegal in C++, a first step to find out is to ask the compiler. However you should be aware of two issues, that we already mentioned when introducing the compiler: First a compiler may have bugs just like any other software, so it happens that not every source code which is forbidden in C++ is properly rejected by the compiler, and vice versa. The other issue is even more dangerous: You can write programs in C++ which a C++ implementation is not required to reject, but whose behavior is not defined by the language. Needless to say, running such a program can, and occasionally will, do harmful things to the system it is running or produce corrupt output!

For example:

```
int MyFunc(); // returns an int
SOMETYPE MyFunc(); // returns a SOMETYPE

int* MyFunc(); // returns a pointer to an int
SOMETYPE *MyFunc(); // returns a pointer to a SOMETYPE
SOMETYPE &MyFunc(); // returns a reference to a SOMETYPE
```

If you have understood the syntax of pointer declarations, the declaration of a function that returns a pointer or a reference should seem logical. The above piece of

code shows how to declare a function that will return a reference or a pointer; below are outlines of what the definitions (implementations) of such functions would look like:

```
SOMETYPE *MyFunc(int *p)
{
  //...

  return p;
}

SOMETYPE &MyFunc(int &r)
{
  //...

  return r;
}
```

return

The `return` statement causes execution to jump from the current function to whatever function called the current function. An optional a result (*return variable*) can be returned. A function may have more than one `return` statement (but returning the same type).

Syntax

```
return;
return value;
```

Within the body of the function, the `return` statement should NOT `return` a pointer or a reference that has the address in memory of a local variable that was declared within the function, because as soon as the function exits, all local variables are destroyed and your pointer or reference will be pointing to some place in memory which you no longer own, so you cannot guarantee its contents. If the object to which a pointer refers is destroyed, the pointer is said to be a *dangling pointer* until it is given a new value; any use of the value of such a pointer is invalid. Having a dangling pointer like that is dangerous; pointers or references to local variables must not be allowed to escape the function in which those local (aka automatic) variables live.

However, within the body of your function, if your pointer or reference has the address in memory of a data type, **struct**, or class that you dynamically allocated the memory for, using the new operator, then returning said pointer or reference would be reasonable:

```
SOMETYPE *MyFunc()  //returning a pointer that has a dynamically
{           //allocated memory address is valid code
  int *p = new int[5];

  //...

  return p;
}
```

In most cases, a better approach in that case would be to `return` an object such as a smart pointer which could manage the memory; explicit memory management using widely distributed calls to `new` and `delete` (or `malloc` and `free`) is tedious, verbose and error prone. At the very least, functions which `return` dynamically allocated resources should be carefully documented. See this book's section on memory management for more details.

```
const SOMETYPE *MyFunc(int *p)
{
  //...

  return p;
}
```

In this case the SOMETYPE object pointed to by the returned pointer may not be modified, and if SOMETYPE is a class then only **const** member functions may be called on the SOMETYPE object.

If such a **const** `return` value is a pointer or a reference to a class then we cannot call non-const methods on that pointer or reference since that would break our agreement not to change it.

Note:
As a general rule methods should be **const** except when it's not possible to make them such. While getting used to the semantics you can use the compiler to inform you when a method may not be **const** -- it will (usually) give an error if you declare a method **const** that needs to be non-const.

Static returns

When a function returns a variable (or a pointer to one) that is statically located, one must keep in mind that it will be possible to overwrite its content each time a function that uses it is called. If you want to save the return value of this function,

you should manually save it elsewhere. Most such static returns use GLOBAL VARIABLES[290].

Of course, when you save it elsewhere, you should make sure to actually copy the value(s) of this variable to another location. If the return value is a struct, you should make a new struct, then copy over the members of the struct.

One example of such a function is the STANDARD C LIBRARY[291] function LOCALTIME[292].

293

Return "codes" (best practices)

There are 2 kinds of behaviors :

> **Note:**
> The selection of, and consistent use of this practice helps to avoid simple errors. Personal taste or organizational dictates may influence the decision, but a general rule-of-thumb is that you should follow whatever choice has been made in the CODE BASE[a] you are currently working in. However, there may be valid reasons for making a different choice in any particular situation.

a HTTP://EN.WIKIPEDIA.ORG/WIKI/CODE_BASE

Positive means success

This is the "logical" way to think, and as such the one used by almost all beginners. In C++, this takes the form of a boolean true/false test, where "true" (also 1 or any non-zero number) means success, and "false" (also 0) means failure.

The major problem of this construct is that all errors return the same value (false), so you must have some kind of externally visible error code in order to determine where the error occurred. For example:

```
bool bOK;
if (my_function1())
{
    // block of instruction 1
```

290 Chapter 3.3.3 on page 137
291 Chapter 3.7.10 on page 264
292 Chapter 3.7.11 on page 349
293 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

```
    if (my_function2())
    {
        // block of instruction 2
        if (my_function3())
        {
            // block of instruction 3
            // Everything worked
            error_code = NO_ERROR;
            bOK = true;
        }
        else
        {
            //error handler for function 3 errors
            error_code = FUNCTION_3_FAILED;
            bOK = false;
        }
    }
    else
    {
        //error handler for function 2 errors
        error_code = FUNCTION_2_FAILED;
        bOK = false;
    }
}
else
{
    //error handler for function 1 errors
    error_code = FUNCTION_1_FAILED;
    bOK = false;
}
return bOK;
```

As you can see, the else blocks (usually error handling) of my_function1 can be really far from the test itself; this is the first problem. When your function begins to grow, it's often difficult to see the test and the error handling at the same time.

This problem can be compensated by SOURCE CODE EDITOR[294] features such as folding, or by testing for a function returning "false" instead of true.

```
if (!my_function1()) // or if (my_function1() == false)
{
    //error handler for function 1 errors

    //...
```

This can also make the code look more like the "0 means success" paradigm, but a little less readable.

The second problem of this construct is that it tends to break up logical tests (my_-function2 is one level more indented, my_function3 is 2 levels indented) which causes legibility problems.

294 HTTP://EN.WIKIPEDIA.ORG/WIKI/SOURCE_CODE_EDITOR

One advantage here is that you follow the STRUCTURED PROGRAMMING[295] principle of a function having a single entry and a single exit.

The MICROSOFT FOUNDATION CLASS LIBRARY[296] (MFC) is an example of a standard library that uses this paradigm.

0 means success

This means that if a function returns 0, the function has completed successfully. Any other value means that an error occurred, and the value returned may be an indication of what error occurred.

The advantage of this paradigm is that the error handling is closer to the test itself. For example the previous code becomes:

```
if (0 != my_function1())
{
    //error handler for function 1 errors
    return FUNCTION_1_FAILED;
}
// block of instruction 1
if (0 != my_function2())
{
    //error handler for function 2 errors
    return FUNCTION_2_FAILED;
}
// block of instruction 2
if (0 != my_function3())
{
    //error handler for function 3 errors
    return FUNCTION_3_FAILED;
}
// block of instruction 3
// Everything worked
return 0; // NO_ERROR
```

In this example, this code is more readable (this will not always be the case). However, this function now has multiple exit points, violating a principle of structured programming.

The C STANDARD LIBRARY[297] (libc) is an example of a standard library that uses this paradigm.

295 HTTP://EN.WIKIPEDIA.ORG/WIKI/STRUCTURED_PROGRAMMING

296 HTTP://EN.WIKIPEDIA.ORG/WIKI/MICROSOFT_FOUNDATION_CLASS_LIBRARY

297 HTTP://EN.WIKIPEDIA.ORG/WIKI/C_STANDARD_LIBRARY

Note:
Some people argue that using functions results in a performance penalty. In this case just use inline functions and let the compiler do the work. Small functions mean visibility, easy debugging and easy maintenance.

3.7.5 Composition

Just as with mathematical functions, C++ functions can be composed, meaning that you use one expression as part of another. For example, you can use any expression as an argument to a function: `double x = cos (angle + pi/2);`

This statement takes the value of pi, divides it by two and adds the result to the value of angle. The sum is then passed as an argument to the cos function.

You can also take the result of one function and pass it as an argument to another: `double x = exp (log (10.0));`

This statement finds the log base e of 10 and then raises e to that power. The result gets assigned to x; I hope you know what it is.

3.7.6 Recursion

In programming languages, RECURSION[298] was first implemented in LISP[299] on the basis of a mathematical concept that existed earlier on, it is a concept that allows us to break down a problem into one or more subproblems that are similar in form to the original problem, in this case, of having a function call itself in some circumstances. It is generally distinguished from ITERATORS OR LOOPS[300].

A simple example of a recursive function is:

```
void func(){
   func();
}
```

It should be noted that non-terminating recursive functions as shown above are almost never used in programs (indeed, some definitions of recursion would exclude

298 HTTP://EN.WIKIPEDIA.ORG/WIKI/RECURSION
299 HTTP://EN.WIKIBOOKS.ORG/WIKI/PROGRAMMING%3ALISP
300 Chapter 3.6.1 on page 214

such non-terminating definitions). A terminating condition is used to prevent infinite recursion.

Example

```
double power(double x, int n)
{
 if (n < 0)
 {
    std::cout << std::endl
              << "Negative index, program terminated.";
    exit(1);
 }
 if (n)
    return x * power(x, n-1);
 else
    return 1.0;
}
```

The above function can be called like this:

```
x = power(x, static_cast<int>(power(2.0, 2)));
```

Why is recursion useful? Although, theoretically, anything possible by recursion is also possible by iteration (that is, `while`), it is sometimes much more convenient to use recursion. Recursive code happens to be much easier to follow as in the example below. The problem with recursive code is that it takes too much memory. Since the function is called many times, without the data from the calling function removed, memory requirements increase significantly. But often the simplicity and elegance of recursive code overrules the memory requirements.

The classic example of recursion is the factorial: $n! = (n-1)!n$, where $0! = 1$ by convention. In recursion, this function can be succinctly defined as

```
unsigned factorial(unsigned n)
{
  if(n != 0)
  {
    return n * factorial(n-1);
  }
  else
  {
    return 1;
  }
}
```

With iteration, the logic is harder to see:

```
unsigned factorial2(unsigned n)
{
  int a = 1;
  while(n > 0)
  {
    a = a*n;
    n = n-1;
  }
  return a;
}
```

Although recursion tends to be slightly slower than iteration, it should be used where using iteration would yield long, difficult-to-understand code. Also, keep in mind that recursive functions take up additional memory (on the stack) for each level. Thus they can run out of memory where an iterative approach may just use constant memory.

Each recursive function needs to have a **Base Case**. A base case is where the recursive function stops calling itself and returns a value. The value returned is (hopefully) the desired value.

For the previous example,

```
unsigned factorial(unsigned n)
{
  if(n != 0)
  {
    return n * factorial(n-1);
  }
  else
  {
    return 1;
  }
}
```

the base case is reached when $n = 0$. In this example, the base case is everything contained in the else statement (which happens to return the number 1). The overall value that is returned is every value from n to 0 multiplied together. So, suppose we call the function and pass it the value 3. The function then does the math $3 * 2 * 1 = 6$ and returns 6 as the result of calling factorial(3).

Another classic example of recursion is the sequence of Fibonacci numbers:

```
0 1 1 2 3 5 8 13 21 34 ...
```

The zeroth element of the sequence is 0. The next element is 1. Any other number of this series is the sum of the two elements coming before it. As an exercise, write a function that returns the *n*th Fibonacci number using recursion.

3.7.7 main

The function `main` also happens to be the *entry point* of any (standard-compliant) C++ program and must be defined. The compiler arranges for the `main` function to be called when the program begins execution. `main` may *call* other functions which may call yet other functions.

Note:
`main` also special because the user code is not allowed to call it; in particular, it cannot be directly or indirectly recursive. This is one of the many small ways in which C++ differs from C.

The `main` function returns an integer value. In certain systems, this value is interpreted as a success/failure code. The return value of zero signifies a successful completion of the program. Any non-zero value is considered a failure. Unlike other functions, if control reaches the end of `main()`, an implicit `return 0;` for success is automatically added. To make return values from `main` more readable, the header file `cstdlib` defines the constants `EXIT_SUCCESS` and `EXIT_FAILURE` (to indicate successful/unsuccessful completion respectively).

Note:
The ISO C++ Standard (ISO/IEC 14882:1998) specifically requires main to have a return type of `int`. But the ISO C Standard (ISO/IEC 9899:1999) actually does not, though most compilers treat this as a minor warning-level error. The explicit use of `return 0;` (or `return EXIT_SUCCESS;`) to exit the main function is left to the CODING STYLE[a] used.

a Chapter 3.1.8 on page 61

The main function can also be declared like this:

```
int main(int argc, char **argv){
  // code
}
```

which defines the `main` function as returning an integer value **int** and taking two parameters. The first parameter of the `main` function, **argc**, is an integer value **int** that specifies the number of arguments passed to the program, while the second, **argv**, is an array of strings containing the actual arguments. There is almost always at least one argument passed to a program; the name of the program itself is the first argument, `argv[0]`. Other arguments may be passed from the system.

Example

```
#include <iostream>

int main(int argc, char **argv){
  std::cout << "Number of arguments: " << argc << std::endl;
  for(size_t i = 0; i < argc; i++)
    std::cout << "  Argument " << i << " = '" << argv[i] << "'" << std::endl;
}
```

Note:
size_t is the return type of `sizeof` function. **size_t** is a **typedef** for some `unsigned` type and is often defined as `unsigned` **int** or `unsigned` **long** but not always.

If the program above is compiled into the executable `arguments` and executed from the command line like this in *nix:

```
$ ./arguments I love chocolate cake
```

Or in Command Prompt in Windows or MS-DOS:

```
C:\>arguments I love chocolate cake
```

It will output the following (but note that argument 0 may not be quite the same as this -- it might include a full path, or it might include the program name only, or it might include a relative path, or it might even be empty):

```
Number of arguments: 5
  Argument 0 = './arguments'
  Argument 1 = 'I'
  Argument 2 = 'love'
  Argument 3 = 'chocolate'
  Argument 4 = 'cake'
```

You can see that the command line arguments of the program are stored into the `argv` array, and that `argc` contains the length of that array. This allows you to change the behavior of a program based on the command line arguments passed to it.

Note:
argv is a (pointer to the first element of an) array of strings. As such, it can be written as `char **argv` or as `char *argv[]`. However, `char argv[][]` is not allowed. Read up on C++ arrays for the exact reasons for this.
Also, **argc** and **argv** are the two most common names for the two arguments given to the `main` function. You can think them to stand for "arguments count" and "arguments variables" respectively. They can, however, be changed if you'd like. The following code is just as legal:

```
int main(int foo, char **bar){ // code }
```

However, any other programmer that sees your code might get mad at you if you code like that.
From the example above, we can also see that C++ do not really care about what the variables' names are (of course, you cannot use reserved words as names) but their types.

3.7.8 Pointers to functions

The POINTERS[301] we have looked at so far have all been data pointers, pointers to functions (more often called function pointers) are very similar and share the same characteristics of other pointers but in place of pointing to a variable they point to functions. Creating an extra level of indirection, as a way to use the FUNCTIONAL PROGRAMMING[302] paradigm in C++, since it facilitates calling functions which are determined at runtime from the same piece of code. They allow passing a function around as parameter or return value in another function.

Using function pointers has exactly the same overhead as any other function call plus the additional pointer indirection and since the function to call is determined only at runtime, the compiler will typically not inline the function call as it could do anywhere else. Because of this characteristics, using function pointers may add up to be significantly slower than using regular function calls, and be avoided as a way to gain performance.

301 Chapter 3.4.10 on page 184
302 HTTP://EN.WIKIPEDIA.ORG/WIKI/FUNCTIONAL%20PROGRAMMING

Note:
Function pointers are mostly used in C, C++ also permits another constructs to enable FUNCTIONAL PROGRAMMING[a] that are called FUNCTORS[b] (class type functors and template type functors) that have some advantages over function pointers.

a HTTP://EN.WIKIPEDIA.ORG/WIKI/FUNCTIONAL%20PROGRAMMING
b HTTP://EN.WIKIPEDIA.ORG/WIKI/FUNCTION%20OBJECT

To declare a pointer to a function naively, the name of the pointer must be parenthesized, otherwise a function returning a pointer will be declared. You also have to declare the function's return type and its parameters. These must be exact!

Consider:

```
int (*ptof)(int arg);
```

The function to be referenced must obviously have the same return type and the same parameter type as that of the pointer to function. The address of the function can be assigned just by using its name, optionally prefixed with the address-of operator &. Calling the function can be done by using either *ptof(<value>)* or *(*ptof)(<value>)*.

So:

```
int (*ptof)(int arg);
int func(int arg){
    //function body
}
ptof = &func; // get a pointer to func
ptof = func;  // same effect as ptof = &func
(*ptof)(5);   // calls func
ptof(5);      // same thing.
```

A function returning a `float` can't be pointed to by a pointer returning a `double`. If two names are identical (such as `int` and `signed`, or a `typedef` name), then the conversion is allowed. Otherwise, they must be entirely the same. You define the pointer by grouping the `*` with the variable name as you would any other pointer. The problem is that it might get interpreted as a return type instead.

It is often clearer to use a typedef for function pointer types; this also provides a place to give a meaningful name to the function pointer's type:

```
typedef int (*int_to_int_function)(int);
int_to_int_function ptof;
```

```
int *func (int);   // WRONG: Declares a function taking an int returning
 pointer-to-int.
int (*func) (int); // RIGHT: Defines a pointer to a function taking an int
 returning int.
```

To help reduce confusion, it is popular to `typedef` either the function type or the pointer type:

```
typedef int ifunc (int);    // now "ifunc" means "function taking an int
 returning int"
typedef int (*pfunc) (int); // now "pfunc" means "pointer to function taking an
 int returning int"
```

If you `typedef` the function type, you can declare, but not define, functions with that type. If you `typdef` the pointer type, you cannot either declare or define functions with that type. Which to use is a matter of style (although the pointer is more popular).

To assign a pointer to a function, you simply assign it to the function name. The `&` operator is optional (it's not ambiguous). The compiler will automatically select an overloaded version of the function appropriate to the pointer, if one exists:

```
int f (int, int);
int f (int, double);
int g (int, int = 4);
double h (int);
int i (int);

int (*p) (int) = &g; // ERROR: The default parameter needs to be included in the
 pointer type.
p = &h;              // ERROR: The return type needs to match exactly.
p = &i;              // Correct.
p = i;               // Also correct.

int (*p2) (int, double);
p2 = f;              // Correct: The compiler automatically picks "int f (int,
 double)".
```

Using a pointer to a function is even simpler - you simply call it like you would a function. You are allowed to dereference it using the `*` operator, but you don't have to:

```
#include <iostream>

int f (int i) { return 2 * i; }

int main ()
 {
  int (*g) (int) = f;
  std::cout<<"g(4) is "<<g(4)<<std::endl;    // Will output "g(4) is 8"
  std::cout<<"(*g)(5) is "<<g(5)<<std::endl; // Will output "g(5) is 10"
```

```
  return 0;
}
```

303

3.7.9 Callback

In COMPUTER PROGRAMMING[304], a **callback** is EXECUTABLE CODE[305] that is passed as an ARGUMENT[306] to other code. It allows a lower-level ABSTARACTION LAYER[307] to call a FUNCTION[308] defined in a higher-level layer. A callback is often back on the level of the original caller.

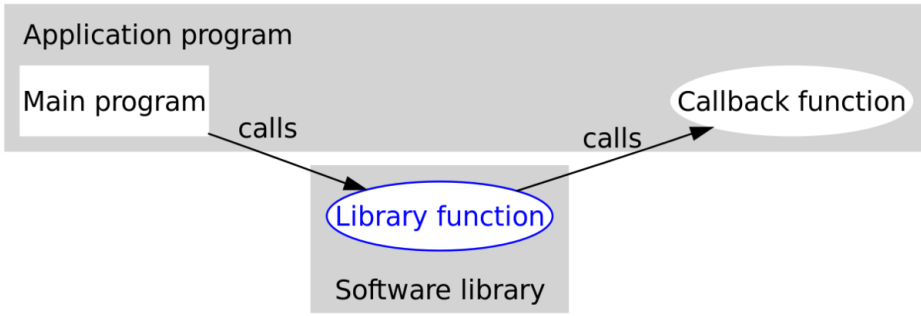


Figure 22: A callback is often back on the level of the original caller.

Usually, the higher-level code starts by calling a function within the lower-level code, passing to it a POINTER[309] or HANDLE[310] to another function. While the lower-level function executes, it may call the passed-in function any number of times to perform some subtask. In another scenario, the lower-level function registers the passed-in function as a *handler* that is to be called asynchronously by the lower-level at a later time in reaction to something.

303 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
304 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPUTER%20PROGRAMMING
305 HTTP://EN.WIKIPEDIA.ORG/WIKI/EXECUTABLE%20CODE
306 HTTP://EN.WIKIPEDIA.ORG/WIKI/ARGUMENT%20%28COMPUTER%20SCIENCE%29
307 HTTP://EN.WIKIPEDIA.ORG/WIKI/ABSTRACTION%20LAYER
308 HTTP://EN.WIKIPEDIA.ORG/WIKI/SUBROUTINE
309 HTTP://EN.WIKIPEDIA.ORG/WIKI/POINTER
310 HTTP://EN.WIKIPEDIA.ORG/WIKI/SMART%20POINTER

A callback can be used as a simpler alternative to POLYMORPHISM[311] and GENERIC PROGRAMMING[312], in that the exact behavior of a function can be dynamically determined by passing different (yet compatible) function pointers or handles to the lower-level function. This can be a very powerful technique for CODE REUSE[313]. In another common scenario, the callback is first registered and later called asynchronously.

311 HTTP://EN.WIKIPEDIA.ORG/WIKI/POLYMORPHISM%20%28COMPUTER%20SCIENCE%29

312 HTTP://EN.WIKIPEDIA.ORG/WIKI/GENERIC%20PROGRAMMING

313 HTTP://EN.WIKIPEDIA.ORG/WIKI/CODE%20REUSE

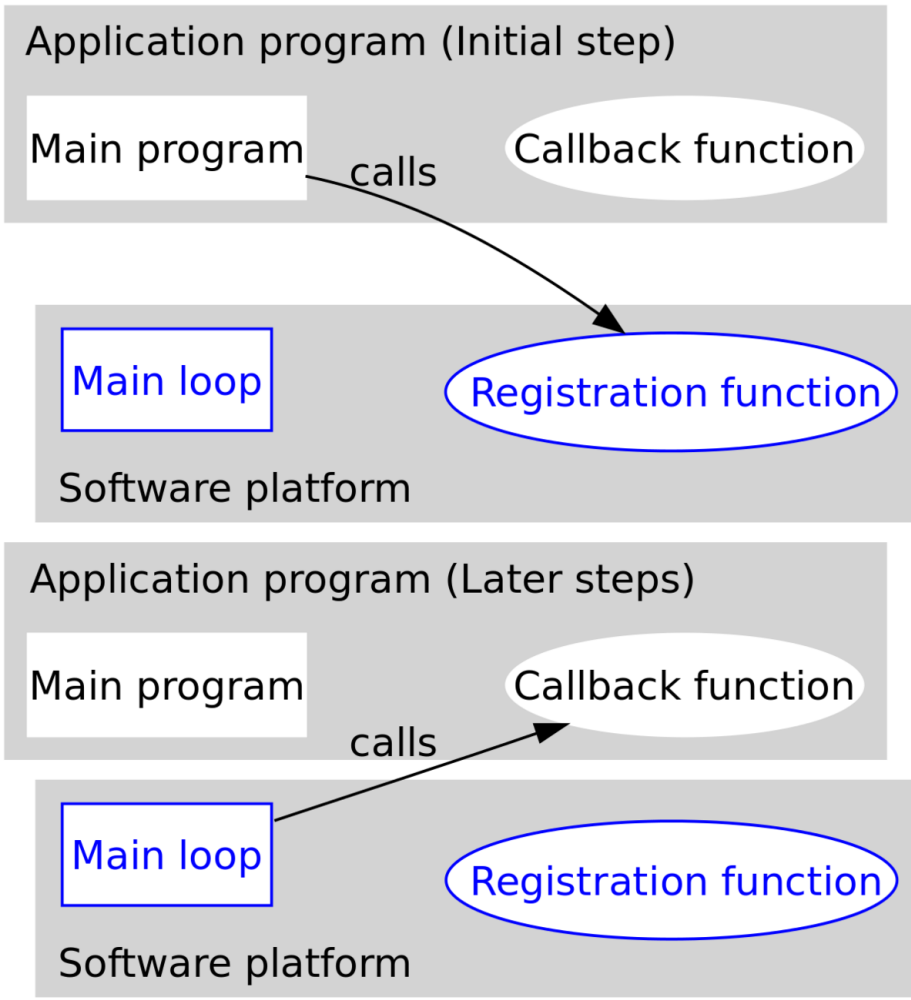


Figure 23: In another common scenario, the callback is first registered and later called asynchronously.

3.7.10 Overloading

Function overloading is the use of a single name for several different functions in the same scope. Multiple functions who share the same name must be differentiated by using another set of parameters for every such function. The functions can be different in the number of parameters they expect, or their parameters can differ

in type. This way, the compiler can figure out the exact function to call by looking at the arguments the caller supplied. This is called overload resolution, and is quite complex.

```
// Overloading Example

// (1)
double geometric_mean( int, int );

// (2)
double geometric_mean( double, double );

// (3)
double geometric_mean( double, double, double );

// ...

// Will call (1):
geometric_mean( 10, 25 );
// Will call (2):
geometric_mean( 22.1, 421.77 );
// Will call (3):
geometric_mean( 11.1, 0.4, 2.224 );
```

Under some circumstances, a call can be ambiguous, because two or more functions match with the supplied arguments equally well.

Example, supposing the declaration of geometric_mean above:

```
// This is an error, because (1) could be called and the second
// argument casted to an int, and (2) could be called with the first
// argument casted to a double. None of the two functions is
// unambiguously a better match.
geometric_mean(7, 13.21);
// This will call (3) too, despite its last argument being an int,
// Because (3) is the only function which can be called with 3
// arguments
geometric_mean(1.1, 2.2, 3);
```

Templates and non-templates can be overloaded. A non-template function takes precedence over a template, if both forms of the function match the supplied arguments equally well.

Note that you can overload many operators in C++ too.

Overloading resolution

Please beware that overload resolution in C++ is one of the most complicated parts of the language. This is probably unavoidable in any case with automatic template

instantiation, user defined implicit conversions, built-in implicit conversation and more as language features. So do not despair if you do not understand this at first go. It is really quite natural, once you have the ideas, but written down it seems extremely complicated.

The easiest way to understand overloading is to imagine that the compiler first finds every function which might possibly be called, using any legal conversions and template instantiations. The compiler then selects the best match, if any, from this set. Specifically, the set is constructed like this:

- All functions with matching name, including function templates, are put into the set. Return types and visibility are not considered. Templates are added with as closely matching parameters as possible. Member functions are considered functions with the first parameter being a pointer-to-class-type.
- Conversion functions are added as so-called surrogate functions, with two parameters, the first being the class type and the second the return type.
- All functions that do not match the number of parameters, even after considering defaulted parameters and ellipses, are removed from the set.
- For each function, each argument is considered to see if a legal conversion sequence exists to convert the caller's argument to the function's parameters. If no such conversion sequence can be found, the function is removed from the set.

The legal conversions are detailed below, but in short a legal conversion is any number of built-in (like int to float) conversions combined with *at most one user defined conversion*. The last part is critical to understand if you are writing replacements to built-in types, such as smart pointers. User defined conversions are described above, but to summarize it is

1. implicit conversion operators like `operator short toShort();`
2. One argument constructors (If a constructor has all but one parameter defaulted, it is considered one-argument)

The overloading resolution works by attempting to establish the best matching function.

Easy conversions are preferred

Looking at one parameter, the preferred conversion is roughly based on scope of the conversion. Specifically, the conversions are preferred in this order, with most-preferred highest:

1. No conversion, adding one or more **const**, adding reference, convert array to pointer to first member

   a) **const** are preferred for rvalues (roughly constants) while non-const are preferred for lvalues (roughly assignables)
2. Conversion from short integral types (**bool**, **char**, **short**) to **int**, and **float** to **double**.
3. Built-in conversions, such as between int and double and pointer type conversion. Pointer conversion are ranked as
   a) Base to derived (pointers) or derived to base (for pointers-to-members), with most-derived preferred
   b) Conversion to `void*`
   c) Conversion to **bool**
4. User-defined conversions, see above.
5. Match with ellipses. (As an aside, this is rather useful knowledge for template meta programming)

The best match is now determined according to the following rules:

- **A function is only a better match if all parameters match at least as well**

In short, the function must be better in every respect --- if one parameter matches better and another worse, neither function is considered a better match. If no function in the set is a better match than both, the call is ambiguous (i.e., it fails) Example:

```
void foo(void*, bool);
void foo(int*, int);

int main() {
   int a;
   foo(&a, true); // ambiguous
}
```

- **Non-templates are preferred over templates**

If all else is equal between two functions, but one is a template and the other not, the non-template is preferred. This seldom causes surprises.

- **Most-specialized template is preferred**

When all else is equal between two template function, but one is more specialized than the other, the most specialized version is preferred. Example:

```
template<typename T> void foo(T);  //1
template<typename T> void foo(T*); //2

int main() {
   int a;
   foo(&a); // Calls 2, since 2 is more specialized.
}
```

Which template is more specialized is an entire chapter unto itself.

- **Return types are ignored**

This rule is mentioned above, but it bears repeating: Return types are *never* part of overload resolutions, even if the function selected has a return type that will cause the compilation to fail. Example:

```
void foo(int);
int foo(float);

int main() {
   // This will fail since foo(int) is best match, and void cannot be converted
to int.
   return foo(5);
}
```

- **The selected function may not be accessible**

If the selected best function is not accessible (e.g., it is a private function and the call it not from a member or friend of its class), the call fails.

314

3.7.11 Standard C Library

The **C standard library** is the C language standardized collection of header files and library routines used to implement common operations, such as input/output

314 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

and string handling. It became part of the C++ STANDARD LIBRARY[315] as the **Standard C Library** in its ANSI C 89 form with some small modifications to make it work better with the C++ Standard Library but remaining outside of the std `namespace`. Header files in the C++ Standard Library do not end in ".h". However, the C++ Standard Library includes 18 header files from the C Standard Library, with ".h" endings. Their use is deprecated (ISO/IEC 14882:2003(E) *Programming Languages — C++*).

For a more in depth look into the C programming language check the C PROGRAMMING WIKIBOOK[316] but be aware of the incompatibilities we have already covered on the COMPARING C++ WITH C SECTION[317] of this book.

All Standard C Library Functions

Functions	Descriptions
ABORT[318]	stops the program
ABS[319]	absolute value
ACOS[320]	arc cosine
ASCTIME[321]	a textual version of the time
ASIN[322]	arc sine
ASSERT[323]	stops the program if an expression isn't true
ATAN[324]	arc tangent
ATAN2[325]	arc tangent, using signs to determine quadrants
ATEXIT[326]	sets a function to be called when the program exits
ATOF[327]	converts a string to a double

315 Chapter 3.1.2 on page 45
316 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%20PROGRAMMING
317 Chapter 2.3.7 on page 25
318 Chapter 3.7.11 on page 356
319 Chapter 3.7.11 on page 330
320 Chapter 3.7.11 on page 331
321 Chapter 3.7.11 on page 345
322 Chapter 3.7.11 on page 331
323 Chapter 3.7.11 on page 357
324 Chapter 3.7.11 on page 332
325 Chapter 3.7.11 on page 333
326 Chapter 3.7.11 on page 357
327 Chapter 3.7.11 on page 304

Functions	Descriptions
ATOI[328]	converts a string to an integer
ATOL[329]	converts a string to a long
BSEARCH[330]	perform a binary search
CALLOC[331]	allocates and clears a two-dimensional chunk of memory
CEIL[332]	the smallest integer not less than a certain value
CLEARERR[333]	clears errors
CLOCK[334]	returns the amount of time that the program has been running
COS[335]	cosine
COSH[336]	hyperbolic cosine
CTIME[337]	returns a specifically formatted version of the time
DIFFTIME[338]	the difference between two times
DIV[339]	returns the quotient and remainder of a division
EXIT[340]	stop the program
EXP[341]	returns "e" raised to a given power
FABS[342]	absolute value for floating-point numbers
FCLOSE[343]	close a file
FEOF[344]	true if at the end-of-file
FERROR[345]	checks for a file error

328 Chapter 3.7.11 on page 304
329 Chapter 3.7.11 on page 305
330 Chapter 3.7.11 on page 358
331 Chapter 3.7.11 on page 353
332 Chapter 3.7.11 on page 333
333 Chapter 3.7.11 on page 274
334 Chapter 3.7.11 on page 346
335 Chapter 3.7.11 on page 334
336 Chapter 3.7.11 on page 334
337 Chapter 3.7.11 on page 347
338 Chapter 3.7.11 on page 348
339 Chapter 3.7.11 on page 335
340 Chapter 3.7.11 on page 358
341 Chapter 3.7.11 on page 336
342 Chapter 3.7.11 on page 336
343 Chapter 3.7.11 on page 274
344 Chapter 3.7.11 on page 275
345 Chapter 3.7.11 on page 275

Functions	Descriptions
FFLUSH[346]	writes the contents of the output buffer
FGETC[347]	get a character from a stream
FGETPOS[348]	get the file position indicator
FGETS[349]	get a string of characters from a stream
FLOOR[350]	returns the largest integer not greater than a given value
FMOD[351]	returns the remainder of a division
FOPEN[352]	open a file
FPRINTF[353]	print formatted output to a file
FPUTC[354]	write a character to a file
FPUTS[355]	write a string to a file
FREAD[356]	read from a file
FREE[357]	returns previously allocated memory to the operating system
FREOPEN[358]	open an existing stream with a different name
FREXP[359]	decomposes a number into scientific notation
FSCANF[360]	read formatted input from a file
FSEEK[361]	move to a specific location in a file
FSETPOS[362]	move to a specific location in a file

346 Chapter 3.7.11 on page 276
347 Chapter 3.7.11 on page 277
348 Chapter 3.7.11 on page 277
349 Chapter 3.7.11 on page 278
350 Chapter 3.7.11 on page 337
351 Chapter 3.7.11 on page 337
352 Chapter 3.7.11 on page 279
353 Chapter 3.7.11 on page 280
354 Chapter 3.7.11 on page 281
355 Chapter 3.7.11 on page 282
356 Chapter 3.7.11 on page 282
357 Chapter 3.7.11 on page 354
358 Chapter 3.7.11 on page 283
359 Chapter 3.7.11 on page 338
360 Chapter 3.7.11 on page 284
361 Chapter 3.7.11 on page 284
362 Chapter 3.7.11 on page 285

Functions	Descriptions
FTELL[363]	returns the current file position indicator
FWRITE[364]	write to a file
GETC[365]	read a character from a file
GETCHAR[366]	read a character from STDIN
GETENV[367]	get environment information about a variable
GETS[368]	read a string from STDIN
GMTIME[369]	returns a pointer to the current Greenwich Mean Time
ISALNUM[370]	true if a character is alphanumeric
ISALPHA[371]	true if a character is alphabetic
ISCNTRL[372]	true if a character is a control character
ISDIGIT[373]	true if a character is a digit
ISGRAPH[374]	true if a character is a graphical character
ISLOWER[375]	true if a character is lowercase
ISPRINT[376]	true if a character is a printing character
ISPUNCT[377]	true if a character is punctuation
ISSPACE[378]	true if a character is a space character
ISUPPER[379]	true if a character is an uppercase character

363 Chapter 3.7.11 on page 286
364 Chapter 3.7.11 on page 286
365 Chapter 3.7.11 on page 287
366 Chapter 3.7.11 on page 288
367 Chapter 3.7.11 on page 359
368 Chapter 3.7.11 on page 288
369 Chapter 3.7.11 on page 348
370 Chapter 3.7.11 on page 306
371 Chapter 3.7.11 on page 306
372 Chapter 3.7.11 on page 307
373 Chapter 3.7.11 on page 308
374 Chapter 3.7.11 on page 308
375 Chapter 3.7.11 on page 309
376 Chapter 3.7.11 on page 310
377 Chapter 3.7.11 on page 310
378 Chapter 3.7.11 on page 311
379 Chapter 3.7.11 on page 311

Functions	Descriptions
ISXDIGIT[380]	true if a character is a hexadecimal character
LABS[381]	absolute value for long integers
LDEXP[382]	computes a number in scientific notation
LDIV[383]	returns the quotient and remainder of a division, in long integer form
LOCALTIME[384]	returns a pointer to the current time
LOG[385]	natural logarithm
LOG10[386]	natural logarithm, in base 10
LONGJMP[387]	start execution at a certain point in the program
MALLOC[388]	allocates memory
MEMCHR[389]	searches an array for the first occurrence of a character
MEMCMP[390]	compares two buffers
MEMCPY[391]	copies one buffer to another
MEMMOVE[392]	moves one buffer to another
MEMSET[393]	fills a buffer with a character
MKTIME[394]	returns the calendar version of a given time
MODF[395]	decomposes a number into integer and fractional parts
PERROR[396]	displays a string version of the current error to STDERR

380 Chapter 3.7.11 on page 312
381 Chapter 3.7.11 on page 338
382 Chapter 3.7.11 on page 339
383 Chapter 3.7.11 on page 339
384 Chapter 3.7.11 on page 349
385 Chapter 3.7.11 on page 340
386 Chapter 3.7.11 on page 341
387 Chapter 3.7.11 on page 359
388 Chapter 3.7.11 on page 354
389 Chapter 3.7.11 on page 312
390 Chapter 3.7.11 on page 313
391 Chapter 3.7.11 on page 314
392 Chapter 3.7.11 on page 314
393 Chapter 3.7.11 on page 315
394 Chapter 3.7.11 on page 349
395 Chapter 3.7.11 on page 341
396 Chapter 3.7.11 on page 289

Functions	Descriptions
POW[397]	returns a given number raised to another number
PRINTF[398]	write formatted output to STDOUT
PUTC[399]	write a character to a stream
PUTCHAR[400]	write a character to STDOUT
PUTS[401]	write a string to STDOUT
QSORT[402]	perform a quicksort
RAISE[403]	send a signal to the program
RAND[404]	returns a pseudo-random number
REALLOC[405]	changes the size of previously allocated memory
REMOVE[406]	erase a file
RENAME[407]	rename a file
REWIND[408]	move the file position indicator to the beginning of a file
SCANF[409]	read formatted input from STDIN
SETBUF[410]	set the buffer for a specific stream
SETJMP[411]	set execution to start at a certain point
SETLOCALE[412]	sets the current locale
SETVBUF[413]	set the buffer and size for a specific stream
SIGNAL[414]	register a function as a signal handler

397 Chapter 3.7.11 on page 342
398 Chapter 3.7.11 on page 290
399 Chapter 3.7.11 on page 293
400 Chapter 3.7.11 on page 294
401 Chapter 3.7.11 on page 294
402 Chapter 3.7.11 on page 360
403 Chapter 3.7.11 on page 361
404 Chapter 3.7.11 on page 361
405 Chapter 3.7.11 on page 355
406 Chapter 3.7.11 on page 295
407 Chapter 3.7.11 on page 295
408 Chapter 3.7.11 on page 296
409 Chapter 3.7.11 on page 296
410 Chapter 3.7.11 on page 298
411 Chapter 3.7.11 on page 362
412 Chapter 3.7.11 on page 350
413 Chapter 3.7.11 on page 299
414 Chapter 3.7.11 on page 363

Functions	Descriptions
SIN[415]	sine
SINH[416]	hyperbolic sine
SPRINTF[417]	write formatted output to a buffer
SQRT[418]	square root
SRAND[419]	initialize the random number generator
SSCANF[420]	read formatted input from a buffer
STRCAT[421]	concatenates two strings
STRCHR[422]	finds the first occurrence of a character in a string
STRCMP[423]	compares two strings
STRCOLL[424]	compares two strings in accordance to the current locale
STRCPY[425]	copies one string to another
STRCSPN[426]	searches one string for any characters in another
STRERROR[427]	returns a text version of a given error code
STRFTIME[428]	returns individual elements of the date and time
STRLEN[429]	returns the length of a given string
STRNCAT[430]	concatenates a certain amount of characters of two strings
STRNCMP[431]	compares a certain amount of characters of two strings

415 Chapter 3.7.11 on page 342
416 Chapter 3.7.11 on page 343
417 Chapter 3.7.11 on page 299
418 Chapter 3.7.11 on page 343
419 Chapter 3.7.11 on page 364
420 Chapter 3.7.11 on page 300
421 Chapter 3.7.11 on page 316
422 Chapter 3.7.11 on page 317
423 Chapter 3.7.11 on page 317
424 Chapter 3.7.11 on page 318
425 Chapter 3.7.11 on page 319
426 Chapter 3.7.11 on page 320
427 Chapter 3.7.11 on page 320
428 Chapter 3.7.11 on page 351
429 Chapter 3.7.11 on page 321
430 Chapter 3.7.11 on page 321
431 Chapter 3.7.11 on page 322

Functions	Descriptions
STRNCPY[432]	copies a certain amount of characters from one string to another
STRPBRK[433]	finds the first location of any character in one string, in another string
STRRCHR[434]	finds the last occurrence of a character in a string
STRSPN[435]	returns the length of a substring of characters of a string
STRSTR[436]	finds the first occurrence of a substring of characters
STRTOD[437]	converts a string to a double
STRTOK[438]	finds the next token in a string
STRTOL[439]	converts a string to a long
STRTOUL[440]	converts a string to an `unsigned` long
STRXFRM[441]	converts a substring so that it can be used by string comparison functions
SYSTEM[442]	perform a system call
TAN[443]	tangent
TANH[444]	hyperbolic tangent
TIME[445]	returns the current calendar time of the system
TMPFILE[446]	return a pointer to a temporary file
TMPNAM[447]	return a unique filename

432 Chapter 3.7.11 on page 322
433 Chapter 3.7.11 on page 323
434 Chapter 3.7.11 on page 324
435 Chapter 3.7.11 on page 324
436 Chapter 3.7.11 on page 325
437 Chapter 3.7.11 on page 326
438 Chapter 3.7.11 on page 326
439 Chapter 3.7.11 on page 327
440 Chapter 3.7.11 on page 328
441 Chapter 3.7.11 on page 328
442 Chapter 3.7.11 on page 365
443 Chapter 3.7.11 on page 344
444 Chapter 3.7.11 on page 345
445 Chapter 3.7.11 on page 352
446 Chapter 3.7.11 on page 301
447 Chapter 3.7.11 on page 301

Functions	Descriptions
TOLOWER[448]	converts a character to lowercase
TOUPPER[449]	converts a character to uppercase
UNGETC[450]	puts a character back into a stream
VA_ARG[451]	use variable length parameter lists
VPRINTF, VFPRINTF, AND VSPRINTF[452]	write formatted output with variable argument lists
VSCANF, VFSCANF, AND VSSCANF[453]	read formatted input with variable argument lists

These routines included on the Standard C Library can be sub divided into:

- STANDARD C I/O[454]
- STANDARD C STRING & CHARACTER[455]
- STANDARD C MATH[456]
- STANDARD C TIME & DATE[457]
- STANDARD C MEMORY[458]
- OTHER STANDARD C FUNCTIONS[459]

460

461

448 Chapter 3.7.11 on page 329
449 Chapter 3.7.11 on page 329
450 Chapter 3.7.11 on page 302
451 Chapter 3.7.11 on page 365
452 Chapter 3.7.11 on page 302
453 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FCODE%2FSTANDARD%20C%20LIBRARY%2FFUNCTIONS%2FVSCANF%2C%20VFSCANF%2C%20AND%20VSSCANF
454 Chapter 3.7.11 on page 273
455 Chapter 3.7.11 on page 303
456 Chapter 3.7.11 on page 330
457 Chapter 3.7.11 on page 345
458 Chapter 3.7.11 on page 353
459 Chapter 3.7.11 on page 356
460 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
461 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

Standard C I/O

The Standard C Library includes routines that are somewhat outdated, but due to the HISTORY OF THE C++ LANGUAGE[462] and its objective to maintain compatibility these are included in the package.

C I/O calls still appear in old code (not only ANSI C 89 but even old C++ code). Its use today may depend on a large number of factors, the age of the code base or the level of complexity of the project or even based on the experience of the programmers. Why use something you are not familiar with if you are proficient in C and in some cases C-style I/O routines are superior to their C++ I/O counterparts, for instance they are more compact and may be are good enough for the simple projects that don't make use of classes.

Note:
If you're learning I/O for the first time you probably should program using the C++ I/O system and not bring legacy I/O systems into the mix. Learn C-style I/O only if you have to.

clearerr

Syntax

```
include <cstdio> void clearerr( FILE *stream );
```

The clearerr function resets the error flags and **EOF** indicator for the given stream. If an error occurs, you can use `perror()` or `strerror()` to figure out which error actually occurred, or read the error from the global variable `errno`.

Related topics

FEOF[463] - FERROR[464] - PERROR[465] - STRERROR[466]

467

462 Chapter 2.1 on page 7
463 Chapter 3.7.11 on page 275
464 Chapter 3.7.11 on page 275
465 Chapter 3.7.11 on page 289
466 Chapter 3.7.11 on page 320
467 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

fclose

Syntax

```
include <cstdio> int fclose( FILE *stream );
```

The function fclose() closes the given file stream, deallocating any buffers associated with that stream. fclose() returns 0 upon success, and **EOF** otherwise.

Related topics

FFLUSH[468] - FOPEN[469] - FREOPEN[470] - SETBUF[471]

472

feof

Syntax

```
include <cstdio> int feof( FILE *stream );
```

The function feof() returns **TRUE** if the end-of-file was reached, or **FALSE** otherwise.

Related topics

CLEARERR[473] - FERROR[474] - GETC[475] - PERROR[476] - PUTC[477]

478

468 Chapter 3.7.11 on page 276
469 Chapter 3.7.11 on page 279
470 Chapter 3.7.11 on page 283
471 Chapter 3.7.11 on page 298
472 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
473 Chapter 3.7.11 on page 274
474 Chapter 3.7.11 on page 275
475 Chapter 3.7.11 on page 287
476 Chapter 3.7.11 on page 289
477 Chapter 3.7.11 on page 293
478 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

ferror

Syntax

```
include <cstdio> int ferror( FILE *stream );
```

The ferror() function looks for errors with stream, returning zero if no errors have occurred, and non-zero if there is an error. In case of an error, use perror() to determine which error has occurred.

Related topics

CLEARERR[479] - FEOF[480] - PERROR[481]

[482]

fflush

Syntax

```
include <cstdio> int fflush( FILE *stream );
```

If the given file stream is an output stream, then fflush() causes the output buffer to be written to the file. If the given stream is of the input type, the behavior of fflush() depends on the library being used (for example, some libraries ignore the operation, others report an error, and others clear pending input).

fflush() is useful when either debugging (for example, if a program segfaults before the buffer is sent to the screen), or it can be used to ensure a partial display of output before a long processing period.

By default, most implementations have *stdout* transmit the buffer at the end of each line, while *stderr* is flushed whenever there is output. This behavior changes if there is a redirection or pipe, where calling fflush(**stdout**) can help maintain the flow of output.

479 Chapter 3.7.11 on page 274
480 Chapter 3.7.11 on page 275
481 Chapter 3.7.11 on page 289
482 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

```
printf( "Before first call\n" );
fflush( stdout );
shady_function();
printf( "Before second call\n" );
fflush( stdout );
dangerous_dereference();
```

Related topics

FCLOSE[483] - FOPEN[484] - FREAD[485] - FWRITE[486] - GETC[487] - PUTC[488]

489

fgetc

Syntax

```
include <cstdio> int fgetc( FILE *stream );
```

The fgetc() function returns the next character from stream, or **EOF** if the end of file is reached or if there is an error.

Related topics

FOPEN[490] - FPUTC[491] - FREAD[492] - FWRITE[493] - GETC[494] - GETCHAR[495] - GETS[496] - PUTC[497]

498

483 Chapter 3.7.11 on page 274
484 Chapter 3.7.11 on page 279
485 Chapter 3.7.11 on page 282
486 Chapter 3.7.11 on page 286
487 Chapter 3.7.11 on page 287
488 Chapter 3.7.11 on page 293
489 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
490 Chapter 3.7.11 on page 279
491 Chapter 3.7.11 on page 281
492 Chapter 3.7.11 on page 282
493 Chapter 3.7.11 on page 286
494 Chapter 3.7.11 on page 287
495 Chapter 3.7.11 on page 288
496 Chapter 3.7.11 on page 288
497 Chapter 3.7.11 on page 293
498 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

fgetpos

Syntax

```
include <cstdio> int fgetpos( FILE *stream, fpos_t *position );
```

The fgetpos() function stores the file position indicator of the given file stream in the given position variable. The position variable is of type fpos_t (which is defined in cstdio) and is an object that can hold every possible position in a FILE. fgetpos() returns zero upon success, and a non-zero value upon failure.

Related topics

FSEEK[499] - FSETPOS[500] - FTELL[501]

502

fgets

Syntax

```
include <cstdio> char *fgets( char *str, int num, FILE *stream );
```

The function `fgets()` reads up to `num` - 1 characters from the given file stream and dumps them into `str`. The string that `fgets()` produces is always null-terminated. `fgets()` will stop when it reaches the end of a line, in which case `str` will contain that newline character. Otherwise, `fgets()` will stop when it reaches `num` - 1 characters or encounters the `EOF` character. `fgets()` returns `str` on success, and `NULL` on an error.

Related topics

499 Chapter 3.7.11 on page 284
500 Chapter 3.7.11 on page 285
501 Chapter 3.7.11 on page 286
502 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

FPUTS[503] - FSCANF[504] - GETS[505] - SCANF[506]

507

fopen

Syntax

```
include <cstdio> FILE *fopen( const char *fname, const char *mode );
```

The fopen() function opens a file indicated by fname and returns a stream associated with that file. If there is an error, fopen() returns **NULL**. mode is used to determine how the file will be treated (i.e. for input, output, etc.)

The mode contains up to three characters. The first character is either "r", "w", or "a", which indicates how the file is opened. A file opened for reading starts allows input from the beginning of the file. For writing, the file is erased. For appending, the file is kept and writing to the file will start at the end. The second character is "b", is an optional flag that opens the file as binary - omitting any conversions from different formats of text. The third character "+" is an optional flag that allows read and write operations on the file (but the file itself is opened in the same way.

Mode	Meaning	Mode	Meaning
"r"	Open a text file for reading	"r+"	Open a text file for read/write
"w"	Create a text file for writing	"w+"	Create a text file for read/write
"a"	Append to a text file	"a+"	Open a text file for read/write
"rb"	Open a binary file for reading	"rb+"	Open a binary file for read-/write

503 Chapter 3.7.11 on page 282
504 Chapter 3.7.11 on page 284
505 Chapter 3.7.11 on page 288
506 Chapter 3.7.11 on page 296
507 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

Mode	Meaning	Mode	Meaning
"wb"	Create a binary file for writing	"wb+"	Create a binary file for read-/write
"ab"	Append to a binary file	"ab+"	Open a binary file for read-/write

An example:

```
int ch;
FILE *input = fopen( "stuff", "r" );
ch = getc( input );
```

Related topics

FCLOSE[508] - FFLUSH[509] - FGETC[510] - FPUTC[511] - FREAD[512] - FREOPEN[513] - FSEEK[514] - FWRITE[515] - GETC[516] - GETCHAR[517] - SETBUF[518]

519

fprintf

Syntax

```
include <cstdio> int fprintf( FILE *stream, const char *format, ... );
```

508 Chapter 3.7.11 on page 274
509 Chapter 3.7.11 on page 276
510 Chapter 3.7.11 on page 277
511 Chapter 3.7.11 on page 281
512 Chapter 3.7.11 on page 282
513 Chapter 3.7.11 on page 283
514 Chapter 3.7.11 on page 284
515 Chapter 3.7.11 on page 286
516 Chapter 3.7.11 on page 287
517 Chapter 3.7.11 on page 288
518 Chapter 3.7.11 on page 298
519 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

The fprintf() function sends information (the arguments) according to the specified format to the file indicated by stream. fprintf() works just like PRINTF[520]() as far as the format goes. The return value of fprintf() is the number of characters outputted, or a negative number if an error occurs. An example:

```
char name[20] = "Mary";
FILE *out;
out = fopen( "output.txt", "w" );
if( out != NULL )
  fprintf( out, "Hello %s\n", name );
```

Related topics

FPUTC[521] - FPUTS[522] - FSCANF[523] - PRINTF[524] - SPRINTF[525]

526

fputc

Syntax

```
include <cstdio> int fputc( int ch, FILE *stream );
```

The function fputc() writes the given character ch to the given output stream. The return value is the character, unless there is an error, in which case the return value is **EOF**.

Related topics

520 Chapter 3.7.11 on page 290
521 Chapter 3.7.11 on page 281
522 Chapter 3.7.11 on page 282
523 Chapter 3.7.11 on page 284
524 Chapter 3.7.11 on page 290
525 Chapter 3.7.11 on page 299
526 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

FGETC[527] - FOPEN[528] - FPRINTF[529] - FREAD[530] - FWRITE[531] - GETC[532] - GETCHAR[533] - PUTC[534]

535

fputs

Syntax

```
include <cstdio> int fputs( const char *str, FILE *stream );
```

The fputs() function writes an array of characters pointed to by str to the given output stream. The return value is non-negative on success, and **EOF** on failure.

Related topics

FGETS[536] - FPRINTF[537] - FSCANF[538] - GETS[539] - GETC[540] - PUTS[541]

542

527 Chapter 3.7.11 on page 277
528 Chapter 3.7.11 on page 279
529 Chapter 3.7.11 on page 280
530 Chapter 3.7.11 on page 282
531 Chapter 3.7.11 on page 286
532 Chapter 3.7.11 on page 287
533 Chapter 3.7.11 on page 288
534 Chapter 3.7.11 on page 293
535 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
536 Chapter 3.7.11 on page 278
537 Chapter 3.7.11 on page 280
538 Chapter 3.7.11 on page 284
539 Chapter 3.7.11 on page 288
540 Chapter 3.7.11 on page 287
541 Chapter 3.7.11 on page 294
542 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

fread

Syntax

```
include <cstdio> int fread( void *buffer, size_t size, size_t num, FILE *stream );
```

The function fread() reads num number of objects (where each object is size bytes) and places them into the array pointed to by buffer. The data comes from the given input stream. The return value of the function is the number of things read. You can use FEOF[543]() or FERROR[544]() to figure out if an error occurs.

Related topics

FFLUSH[545] - FGETC[546] - FOPEN[547] - FPUTC[548] - FSCANF[549] - FWRITE[550] - GETC[551]

552

freopen

Syntax

```
include <cstdio> FILE *freopen( const char *fname, const char *mode, FILE *stream );
```

The freopen() function is used to reassign an existing stream to a different file and mode. After a call to this function, the given file stream will refer to fname with

543 Chapter 3.7.11 on page 275
544 Chapter 3.7.11 on page 275
545 Chapter 3.7.11 on page 276
546 Chapter 3.7.11 on page 277
547 Chapter 3.7.11 on page 279
548 Chapter 3.7.11 on page 281
549 Chapter 3.7.11 on page 284
550 Chapter 3.7.11 on page 286
551 Chapter 3.7.11 on page 287
552 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

access given by mode. The return value of freopen() is the new stream, or **NULL** if there is an error.

Related topics

FCLOSE[553] - FOPEN[554]

555

fscanf

Syntax

```
include <cstdio> int fscanf( FILE *stream, const char *format, ... );
```

The function fscanf() reads data from the given file stream in a manner exactly like scanf(). The return value of fscanf() is the number of variables that are actually assigned values, including zero if there were no matches. **EOF** is returned if there was an error reading before the first match.

Related topics

FGETS[556] - FPRINTF[557] - FPUTS[558] - FREAD[559] - FWRITE[560] - SCANF[561] - SSCANF[562]

563

553 Chapter 3.7.11 on page 274
554 Chapter 3.7.11 on page 279
555 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
556 Chapter 3.7.11 on page 278
557 Chapter 3.7.11 on page 280
558 Chapter 3.7.11 on page 282
559 Chapter 3.7.11 on page 282
560 Chapter 3.7.11 on page 286
561 Chapter 3.7.11 on page 296
562 Chapter 3.7.11 on page 300
563 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

fseek

Syntax

```
include <cstdio> int fseek( FILE *stream, long offset, int origin );
```

The function fseek() sets the file position data for the given stream. The origin value should have one of the following values (defined in cstdio):

Name	Explanation
SEEK_SET	Seek from the start of the file
SEEK_CUR	Seek from the current location
SEEK_END	Seek from the end of the file

fseek() returns zero upon success, non-zero on failure. You can use fseek() to move beyond a file, but not before the beginning. Using fseek() clears the EOF flag associated with that stream.

Related topics

FGETPOS[564] - FOPEN[565] - FSETPOS[566] - FTELL[567] - REWIND[568]

569

fsetpos

Syntax

```
include <cstdio> int fsetpos( FILE *stream, const fpos_t *position );
```

564 Chapter 3.7.11 on page 277
565 Chapter 3.7.11 on page 279
566 Chapter 3.7.11 on page 285
567 Chapter 3.7.11 on page 286
568 Chapter 3.7.11 on page 296
569 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

The fsetpos() function moves the file position indicator for the given stream to a location specified by the position object. fpos_t is defined in cstdio. The return value for fsetpos() is zero upon success, non-zero on failure.

Related topics

FGETPOS[570] - FSEEK[571] - FTELL[572]

573

ftell

Syntax

```
include <cstdio> long ftell( FILE *stream );
```

The ftell() function returns the current file position for stream, or -1 if an error occurs.

Related topics

FGETPOS[574] - FSEEK[575] - FSETPOS[576]

577

fwrite

Syntax

```
include <cstdio> int fwrite( const void *buffer, size_t size, size_t count, FILE
*stream );
```

570 Chapter 3.7.11 on page 277
571 Chapter 3.7.11 on page 284
572 Chapter 3.7.11 on page 286
573 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
574 Chapter 3.7.11 on page 277
575 Chapter 3.7.11 on page 284
576 Chapter 3.7.11 on page 285
577 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

The fwrite() function writes, from the array buffer, count objects of size size to stream. The return value is the number of objects written.

Related topics

FFLUSH[578] - FGETC[579] - FOPEN[580] - FPUTC[581] - FREAD[582] - FSCANF[583] - GETC[584] - [585]

getc

Syntax

```
include <cstdio> int getc( FILE *stream );
```

The getc() function returns the next character from stream, or **EOF** if the end of file is reached. getc() is identical to FGETC[586](). For example:

```
int ch;
FILE *input = fopen( "stuff", "r" );

ch = getc( input );
while( ch != EOF ) {
  printf( "%c", ch );
  ch = getc( input );
}
```

Related topics

578 Chapter 3.7.11 on page 276
579 Chapter 3.7.11 on page 277
580 Chapter 3.7.11 on page 279
581 Chapter 3.7.11 on page 281
582 Chapter 3.7.11 on page 282
583 Chapter 3.7.11 on page 284
584 Chapter 3.7.11 on page 287
585 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
586 Chapter 3.7.11 on page 277

FEOF[587] - FFLUSH[588] - FGETC[589] - FOPEN[590] - FPUTC[591] - FGETC[592] - FREAD[593] - FWRITE[594] - PUTC[595] - UNGETC[596]

597

getchar

Syntax

```
include <cstdio> int getchar( void );
```

The getchar() function returns the next character from **stdin**, or **EOF** if the end of file is reached.

Related topics

FGETC[598] - FOPEN[599] - FPUTC[600] - PUTC[601]

602

587 Chapter 3.7.11 on page 275
588 Chapter 3.7.11 on page 276
589 Chapter 3.7.11 on page 277
590 Chapter 3.7.11 on page 279
591 Chapter 3.7.11 on page 281
592 Chapter 3.7.11 on page 277
593 Chapter 3.7.11 on page 282
594 Chapter 3.7.11 on page 286
595 Chapter 3.7.11 on page 293
596 Chapter 3.7.11 on page 302
597 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
598 Chapter 3.7.11 on page 277
599 Chapter 3.7.11 on page 279
600 Chapter 3.7.11 on page 281
601 Chapter 3.7.11 on page 293
602 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

gets

Syntax

```
include <cstdio> char *gets( char *str );
```

The gets() function reads characters from stdin and loads them into str, until a newline or **EOF** is reached. The newline character is translated into a null termination. The return value of gets() is the read-in string, or **NULL** if there is an error.

Note:
gets() does not perform bounds checking, and thus risks overrunning str. For a similar (and safer) function that includes bounds checking, see FGETS[a]().

a Chapter 3.7.11 on page 278

Related topics

FGETC[603] - FGETS[604] - FPUTS[605] - PUTS[606]

607

perror

Syntax

```
include <cstdio> void perror( const char *str );
```

The perror() function writes `str`, a ":" followed by a space, an implementation-defined and/or language-dependent error message corresponding to the global variable errno, and a newline to stderr. For example:

```
char* input_filename = "not_found.txt";
```

603 Chapter 3.7.11 on page 277
604 Chapter 3.7.11 on page 278
605 Chapter 3.7.11 on page 282
606 Chapter 3.7.11 on page 294
607 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

```
FILE* input = fopen( input_filename, "r" );
if( input == NULL ) {
  char error_msg[255];
  sprintf( error_msg, "Error opening file '%s'", input_filename );
  perror( error_msg );
  exit( -1 );
}
```

If the file called not_found.txt is not found, this code will produce the following output:

```
Error opening file 'not_found.txt': No such file or directory
```

If "str" is a null pointer or points to the null byte, only the error message corresponding to errno and a newline are written to stderr.

Related topics

CLEARERR[608] - FEOF[609] - FERROR[610]

611

printf

Syntax

```
include <cstdio> int printf( const char *format, ... );
```

The printf() function prints output to **stdout**, according to format and other arguments passed to printf(). The string format consists of two types of items - characters that will be printed to the screen, and format commands that define how the other arguments to printf() are displayed. Basically, you specify a format string that has text in it, as well as "special" characters that map to the other arguments of printf(). For example, this code

```
char name[20] = "Bob";
```

608 Chapter 3.7.11 on page 274
609 Chapter 3.7.11 on page 275
610 Chapter 3.7.11 on page 275
611 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

```
int age = 21;
printf( "Hello %s, you are %d years old\n", name, age );
```

displays the following output:

```
Hello Bob, you are 21 years old
```

The %s means, "insert the first argument, a string, right here." The %d indicates that the second argument (an integer) should be placed there. There are different %-codes for different variable types, as well as options to limit the length of the variables and whatnot.

Control Character	**Explanation**
%c	a single character
%d	a decimal integer
%i	an integer
%e	scientific notation, with a lowercase "e"
%E	scientific notation, with a uppercase "E"
%f	a floating-point number
%g	use %e or %f, whichever is shorter
%G	use %E or %f, whichever is shorter
%o	an octal number
%x	`unsigned` hexadecimal, with lowercase letters
%X	`unsigned` hexadecimal, with uppercase letters
%u	an `unsigned` integer
%s	a string
%x	a hexadecimal number
%p	a pointer
%n	the argument shall be a pointer to an integer into which is placed the number of characters written so far
%%	a percent sign

A field-length specifier may appear before the final control character to indicate the width of the field:

- **h**, when inserted inside %d, causes the argument to be a short int.
- **l**, when inserted inside %d, causes the argument to be a long.
- **l**, when inserted inside %f, causes the argument to be a double.
- **L**, when inserted inside %d or %f, causes the argument to be a long long or long double respecively.

An integer placed between a % sign and the format command acts as a minimum field width specifier, and pads the output with spaces or zeros to make it long enough. If you want to pad with zeros, place a zero before the minimum field width specifier:

```
%012d
```

You can also include a precision modifier, in the form of a .N where N is some number, before the format command:

```
%012.4d
```

The precision modifier has different meanings depending on the format command being used:

- With %e, %E, and %f, the precision modifier lets you specify the number of decimal places desired. For example, %12.6f will display a floating number at least 12 digits wide, with six decimal places.
- With %g and %G, the precision modifier determines the maximum number of significant digits displayed.
- With %s, the precision modifier simply acts as a maximum field length, to complement the minimum field length that precedes the period.

All of printf()'s output is right-justified, unless you place a minus sign right after the % sign. For example,

```
%-12.4f
```

will display a floating point number with a minimum of 12 characters, 4 decimal places, and left justified. You may modify the %d, %i, %o, %u, and %x type specifiers with the letter l and the letter h to specify long and short data types (e.g. %hd means a short integer). The %e, %f, and %g type specifiers can have the letter l before them to indicate that a double follows. The %g, %f, and %e type specifiers can be preceded with the character '#' to ensure that the decimal point

will be present, even if there are no decimal digits. The use of the '#' character with the %x type specifier indicates that the hexidecimal number should be printed with the '0x' prefix. The use of the '#' character with the %o type specifier indicates that the octal value should be displayed with a 0 prefix.

Inserting a plus sign '+' into the type specifier will force positive values to be preceded by a '+' sign. Putting a space character ' ' there will force positive values to be preceded by a single space character.

You can also include constant escape sequences in the output string.

The return value of printf() is the number of characters printed, or a negative number if an error occurred.

Related topics

FPRINTF[612] - PUTS[613] - SCANF[614] - SPRINTF[615]

616

putc

Syntax

```
include <cstdio> int putc( int ch, FILE *stream );
```

The putc() function writes the character ch to stream. The return value is the character written, or **EOF** if there is an error. For example:

```
int ch;
FILE *input, *output;
input = fopen( "tmp.c", "r" );
output = fopen( "tmpCopy.c", "w" );
ch = getc( input );
while( ch != EOF ) {
  putc( ch, output );
  ch = getc( input );
```

612 Chapter 3.7.11 on page 280
613 Chapter 3.7.11 on page 294
614 Chapter 3.7.11 on page 296
615 Chapter 3.7.11 on page 299
616 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

```
}
fclose( input );
fclose( output );
```

Generates a copy of the file tmp.c called tmpCopy.c.

Related topics

FEOF[617] - FFLUSH[618] - FGETC[619] - FPUTC[620] - GETC[621] - GETCHAR[622] - PUTCHAR[623] - PUTS[624]

625

putchar

Syntax

```
include <cstdio> int putchar( int ch );
```

The putchar() function writes ch to **stdout**. The code

```
putchar( ch );
```

is the same as

```
putc( ch, stdout );
```

The return value of putchar() is the written character, or **EOF** if there is an error.

Related topics

PUTC[626]

627

617 Chapter 3.7.11 on page 275
618 Chapter 3.7.11 on page 276
619 Chapter 3.7.11 on page 277
620 Chapter 3.7.11 on page 281
621 Chapter 3.7.11 on page 287
622 Chapter 3.7.11 on page 288
623 Chapter 3.7.11 on page 294
624 Chapter 3.7.11 on page 294
625 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
626 Chapter 3.7.11 on page 293
627 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

puts

Syntax

```
include <cstdio> int puts( char *str );
```

The function puts() writes str to **stdout**. puts() returns non-negative on success, or **EOF** on failure.

Related topics

FPUTS[628] - GETS[629] - PRINTF[630] - PUTC[631]

632

remove

Syntax

```
include <cstdio> int remove( const char *fname );
```

The remove() function erases the file specified by fname. The return value of remove() is zero upon success, and non-zero if there is an error.

Related topics

RENAME[633]

634

628 Chapter 3.7.11 on page 282
629 Chapter 3.7.11 on page 288
630 Chapter 3.7.11 on page 290
631 Chapter 3.7.11 on page 293
632 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
633 Chapter 3.7.11 on page 295
634 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

rename

Syntax

```
include <cstdio> int rename( const char *oldfname, const char *newfname );
```

The function rename() changes the name of the file *oldfname* to *newfname*. The return value of rename() is zero upon success, non-zero on error.

Related topics

REMOVE[635]

636

rewind

Syntax

```
include <cstdio> void rewind( FILE *stream );
```

The function rewind() moves the file position indicator to the beginning of the specified stream, also clearing the error and **EOF** flags associated with that stream.

Related topics

FSEEK[637]

638

635 Chapter 3.7.11 on page 295
636 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
637 Chapter 3.7.11 on page 284
638 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

scanf

Syntax

```
include <cstdio> int scanf( const char *format, ... );
```

The scanf() function reads input from **stdin**, according to the given format, and stores the data in the other arguments. It works a lot like PRINTF[639](). The format string consists of control characters, whitespace characters, and non-whitespace characters. The control characters are preceded by a % sign, and are as follows:

Control Character	Explanation
%c	a single character
%d	a decimal integer
%i	an integer
%e, %f, %g	a floating-point number
%lf	a double
%o	an octal number
%s	a string
%x	a hexadecimal number
%p	a pointer
%n	an integer equal to the number of characters read so far
%u	an `unsigned` integer
%[]	a set of characters
%%	a percent sign

scanf() reads the input, matching the characters from format. When a control character is read, it puts the value in the next variable. Whitespace (tabs, spaces, etc.) are skipped. Non-whitespace characters are matched to the input, then discarded. If a number comes between the % sign and the control character, then only that many characters will be converted into the variable. If scanf() encounters a set of characters, denoted by the %[] control character, then any characters found within the brackets are read into the variable. The return value of scanf() is the number of variables that were successfully assigned values, or **EOF** if there is an error.

639 Chapter 3.7.11 on page 290

This code snippet uses scanf() to read an int, float, and a double from the user. Note that the variable arguments to scanf() are passed in by address, as denoted by the ampersand (&) preceding each variable:

```
int i;
float f;
double d;

printf( "Enter an integer: " );
scanf( "%d", &i );

printf( "Enter a float: " );
scanf( "%f", &f );

printf( "Enter a double: " );
scanf( "%lf", &d );

printf( "You entered %d, %f, and %f\n", i, f, d );
```

Related topics

FGETS[640] - FSCANF[641] - PRINTF[642] - SSCANF[643]

644

setbuf

Syntax

```
include <cstdio> void setbuf( FILE *stream, char *buffer );
```

The setbuf() function sets stream to use buffer, or, if buffer is **NULL**, turns off buffering. This function expects that the buffer be **BUFSIZ** characters long - since this function does not support specifying the size of the buffer, buffers larger than **BUFSIZ** will be partly unused.

Related topics

640 Chapter 3.7.11 on page 278
641 Chapter 3.7.11 on page 284
642 Chapter 3.7.11 on page 290
643 Chapter 3.7.11 on page 300
644 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

FCLOSE[645] - FOPEN[646] - SETVBUF[647]

648

setvbuf

Syntax

```
include <cstdio> int setvbuf( FILE *stream, char *buffer, int mode, size_t size );
```

The function setvbuf() sets the buffer for *stream* to be *buffer*, with a size of size. mode can be one of:

- _IOFBF, which indicates full buffering
- _IOLBF, which means line buffering
- _IONBF, which means no buffering

Related topics

FFLUSH[649] - SETBUF[650]

651

sprintf

Syntax

```
include <cstdio> int sprintf( char *buffer, const char *format, ... );
```

645 Chapter 3.7.11 on page 274
646 Chapter 3.7.11 on page 279
647 Chapter 3.7.11 on page 299
648 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
649 Chapter 3.7.11 on page 276
650 Chapter 3.7.11 on page 298
651 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

The sprintf() function is just like PRINTF[652](), except that the output is sent to *buffer*. The return value is the number of characters written. For example:

```
char string[50];
int file_number = 0;

sprintf( string, "file.%d", file_number );
file_number++;
output_file = fopen( string, "w" );
```

Note that sprintf() does the opposite of a function like ATOI[653]() -- where ATOI[654]() converts a string into a number, sprintf() can be used to convert a number into a string.

For example, the following code uses sprintf() to convert an integer into a string of characters:

```
char result[100];
int num = 24;
sprintf( result, "%d", num );
```

This code is similar, except that it converts a floating-point number into an array of characters:

```
char result[100];
float fnum = 3.14159;
sprintf( result, "%f", fnum );
```

Related topics

FPRINTF[655] - PRINTF[656]
(Standard C String and Character) ATOF[657] - ATOI[658] - ATOL[659]

660

652 Chapter 3.7.11 on page 290
653 Chapter 3.7.11 on page 304
654 Chapter 3.7.11 on page 304
655 Chapter 3.7.11 on page 280
656 Chapter 3.7.11 on page 290
657 Chapter 3.7.11 on page 304
658 Chapter 3.7.11 on page 304
659 Chapter 3.7.11 on page 305
660 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

sscanf

Syntax

```
include <cstdio> int sscanf( const char *buffer, const char *format, ... );
```

The function sscanf() is just like SCANF[661](), except that the input is read from *buffer*.

Related topics

FSCANF[662] - SCANF[663]

664

tmpfile

Syntax

```
include <cstdio> FILE *tmpfile( void );
```

The function tmpfile() opens a temporary file with a unique filename and returns a pointer to that file. If there is an error, null is returned.

Related topics

TMPNAM[665]

666

661 Chapter 3.7.11 on page 296
662 Chapter 3.7.11 on page 284
663 Chapter 3.7.11 on page 296
664 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
665 Chapter 3.7.11 on page 301
666 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

tmpnam

Syntax

```
include <cstdio> char *tmpnam( char *name );
```

The tmpnam() function creates a unique filename and stores it in name. tmpnam() can be called up to **TMP_MAX** times.

Related topics

TMPFILE[667]

668

ungetc

Syntax

```
include <cstdio> int ungetc( int ch, FILE *stream );
```

The function ungetc() puts the character *ch* back in *stream*.

Related topics

GETC[669]
(C++ I/O) PUTBACK[670]

671

667 Chapter 3.7.11 on page 301
668 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
669 Chapter 3.7.11 on page 287
670 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FCODE%2FIO%2FFUNCTIONS%2FPUTBACK
671 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

vprintf, vfprintf, and vsprintf

Syntax

```
include <cstdarg> include <cstdio> int vprintf( char *format, va_list
arg_ptr ); int vfprintf( FILE *stream, const char *format, va_list arg_-
ptr ); int vsprintf( char *buffer, char *format, va_list arg_ptr );
```

These functions are very much like PRINTF[672](), FPRINTF[673](), and SPRINTF[674](). The difference is that the argument list is a pointer to a list of arguments. **va_list** is defined in cstdarg, and is also used by (Other Standard C Functions) VA_ARG[675]().

For example:

```
void error( char *fmt, ... ) {
  va_list args;
  va_start( args, fmt );
  fprintf( stderr, "Error: " );
  vfprintf( stderr, fmt, args );
  fprintf( stderr, "\n" );
  va_end( args );
  exit( 1 );
}
```

676

677

Standard C String & Character

The Standard C Library includes also routines that deals with characters and strings. You must keep in mind that in C, a string of characters is stored in successive elements of a character array and terminated by the **NULL** character.

```
/* "Hello" is stored in a character array */
char note[SIZE];
```

672 Chapter 3.7.11 on page 290
673 Chapter 3.7.11 on page 280
674 Chapter 3.7.11 on page 299
675 Chapter 3.7.11 on page 365
676 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
677 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

```
note[0] = 'H'; note[1] = 'e'; note[2] = 'l'; note[3] = 'l'; note[4] = 'o';
 note[5] = '\0';
```

Even if outdated this C string and character functions still appear in old code and more so than the previous I/O functions.

atof

Syntax

```
include <cstdlib> double atof( const char *str );
```

The function atof() converts str into a double, then returns that value. str must start with a valid number, but can be terminated with any non-numerical character, other than "E" or "e". For example,

```
x = atof( "42.0is_the_answer" );
```

results in x being set to 42.0.

Related topics

ATOI[678] - ATOL[679] - STRTOD[680]
(Standard C I/O) SPRINTF[681]

682

atoi

Syntax

```
include <cstdlib> int atoi( const char *str );
```

678 Chapter 3.7.11 on page 304
679 Chapter 3.7.11 on page 305
680 Chapter 3.7.11 on page 326
681 Chapter 3.7.11 on page 299
682 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

The atoi() function converts str into an integer, and returns that integer. *str* should start with a whitespace or some sort of number, and atoi() will stop reading from *str* as soon as a non-numerical character has been read. For example:

```
int i;
i = atoi( "512" );
i = atoi( "512.035" );
i = atoi( "   512.035" );
i = atoi( "   512+34" );
i = atoi( "   512 bottles of beer on the wall" );
```

All five of the above assignments to the variable *i* would result in it being set to 512.

If the conversion cannot be performed, then atoi() will return zero:

```
int i = atoi( " does not work: 512" );  // results in i == 0
```

Related topics

ATOF[683] - ATOL[684]
(Standard C I/O) SPRINTF[685]

686

atol

Syntax

```
include <cstdlib> long atol( const char *str );
```

The function atol() converts *str* into a long, then returns that value. atol() will read from *str* until it finds any character that should not be in a long. The resulting truncated value is then converted and returned. For example,

```
x = atol( "1024.0001" );
```

results in x being set to 1024L.

683 Chapter 3.7.11 on page 304
684 Chapter 3.7.11 on page 305
685 Chapter 3.7.11 on page 299
686 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

Related topics

ATOF[687] - ATOI[688] - STRTOD[689]
(Standard C I/O) SPRINTF[690]

691

isalnum

Syntax

```
include <cctype> int isalnum( int ch );
```

The function isalnum() returns non-zero if its argument is a numeric digit or a letter of the alphabet. Otherwise, zero is returned.

```
char c;
scanf( "%c", &c );
if( isalnum(c) )
  printf( "You entered the alphanumeric character %c\n", c );
```

Related topics

ISALPHA[692] - ISCNTRL[693] - ISDIGIT[694] - ISGRAPH[695] - ISPRINT[696] - ISPUNCT[697] - ISSPACE[698] - ISXDIGIT[699]

700

687 Chapter 3.7.11 on page 304
688 Chapter 3.7.11 on page 304
689 Chapter 3.7.11 on page 326
690 Chapter 3.7.11 on page 299
691 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
692 Chapter 3.7.11 on page 306
693 Chapter 3.7.11 on page 307
694 Chapter 3.7.11 on page 308
695 Chapter 3.7.11 on page 308
696 Chapter 3.7.11 on page 310
697 Chapter 3.7.11 on page 310
698 Chapter 3.7.11 on page 311
699 Chapter 3.7.11 on page 312
700 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

isalpha

Syntax

```
include <cctype> int isalpha( int ch );
```

The function isalpha() returns non-zero if its argument is a letter of the alphabet. Otherwise, zero is returned.

```
char c;
scanf( "%c", &c );
if( isalpha(c) )
  printf( "You entered a letter of the alphabet\n" );
```

Related topics

ISALNUM[701] - ISCNTRL[702] - ISDIGIT[703] - ISGRAPH[704] - ISPRINT[705] - ISPUNCT[706] - ISSPACE[707] - ISXDIGIT[708]

709

iscntrl

Syntax

```
include <cctype> int iscntrl( int ch );
```

The iscntrl() function returns non-zero if its argument is a control character (between 0 and 0x1F or equal to 0x7F). Otherwise, zero is returned.

Related topics

701 Chapter 3.7.11 on page 306
702 Chapter 3.7.11 on page 307
703 Chapter 3.7.11 on page 308
704 Chapter 3.7.11 on page 308
705 Chapter 3.7.11 on page 310
706 Chapter 3.7.11 on page 310
707 Chapter 3.7.11 on page 311
708 Chapter 3.7.11 on page 312
709 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

ISALNUM[710] - ISALPHA[711] - ISDIGIT[712] - ISGRAPH[713] - ISPRINT[714] - ISPUNCT[715] - ISSPACE[716] - ISXDIGIT[717]

718

isdigit

Syntax

```
include <cctype> int isdigit( int ch );
```

The function isdigit() returns non-zero if its argument is a digit between 0 and 9. Otherwise, zero is returned.

```
char c;
scanf( "%c", &c );
if( isdigit(c) )
  printf( "You entered the digit %c\n", c );
```

Related topics

ISALNUM[719] - ISALPHA[720] - ISCNTRL[721] - ISGRAPH[722] - ISPRINT[723] - ISPUNCT[724] - ISSPACE[725] - ISXDIGIT[726]

727

---

710 Chapter 3.7.11 on page 306
711 Chapter 3.7.11 on page 306
712 Chapter 3.7.11 on page 308
713 Chapter 3.7.11 on page 308
714 Chapter 3.7.11 on page 310
715 Chapter 3.7.11 on page 310
716 Chapter 3.7.11 on page 311
717 Chapter 3.7.11 on page 312
718 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
719 Chapter 3.7.11 on page 306
720 Chapter 3.7.11 on page 306
721 Chapter 3.7.11 on page 307
722 Chapter 3.7.11 on page 308
723 Chapter 3.7.11 on page 310
724 Chapter 3.7.11 on page 310
725 Chapter 3.7.11 on page 311
726 Chapter 3.7.11 on page 312
727 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

isgraph

Syntax

```
include <cctype> int isgraph( int ch );
```

The function isgraph() returns non-zero if its argument is any printable character other than a space (if you can see the character, then isgraph() will return a non-zero value). Otherwise, zero is returned.

Related topics

ISALNUM[728] - ISALPHA[729] - ISCNTRL[730] - ISDIGIT[731] - ISPRINT[732] - ISPUNCT[733] - ISSPACE[734] - ISXDIGIT[735]

736

islower

Syntax

```
include <cctype> int islower( int ch );
```

The islower() function returns non-zero if its argument is a lowercase letter. Otherwise, zero is returned.

Related topics

ISUPPER[737]

728 Chapter 3.7.11 on page 306
729 Chapter 3.7.11 on page 306
730 Chapter 3.7.11 on page 307
731 Chapter 3.7.11 on page 308
732 Chapter 3.7.11 on page 310
733 Chapter 3.7.11 on page 310
734 Chapter 3.7.11 on page 311
735 Chapter 3.7.11 on page 312
736 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
737 Chapter 3.7.11 on page 311

738

isprint

Syntax

```
include <cctype> int isprint( int ch );
```

The function isprint() returns non-zero if its argument is a printable character (including a space). Otherwise, zero is returned.

Related topics

ISALNUM[739] - ISALPHA[740] - ISCNTRL[741] - ISDIGIT[742] - ISGRAPH[743] - ISPUNCT[744] - ISSPACE[745]

746

ispunct

Syntax

```
include <cctype> int ispunct( int ch );
```

The ispunct() function returns non-zero if its argument is a printing character but neither alphanumeric nor a space. Otherwise, zero is returned.

Related topics

738 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
739 Chapter 3.7.11 on page 306
740 Chapter 3.7.11 on page 306
741 Chapter 3.7.11 on page 307
742 Chapter 3.7.11 on page 308
743 Chapter 3.7.11 on page 308
744 Chapter 3.7.11 on page 310
745 Chapter 3.7.11 on page 311
746 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

ISALNUM[747] - ISALPHA[748] - ISCNTRL[749] - ISDIGIT[750] - ISGRAPH[751] - ISSPACE[752] - ISXDIGIT[753]

754

isspace

Syntax

```
include <cctype> int isspace( int ch );
```

The isspace() function returns non-zero if its argument is some sort of space (i.e. single space, tab, vertical tab, form feed, carriage return, or newline). Otherwise, zero is returned.

Related topics

ISALNUM[755] - ISALPHA[756] - ISCNTRL[757] - ISDIGIT[758] - ISGRAPH[759] - ISPRINT[760] - ISPUNCT[761] - ISXDIGIT[762]

763

747 Chapter 3.7.11 on page 306
748 Chapter 3.7.11 on page 306
749 Chapter 3.7.11 on page 307
750 Chapter 3.7.11 on page 308
751 Chapter 3.7.11 on page 308
752 Chapter 3.7.11 on page 311
753 Chapter 3.7.11 on page 312
754 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
755 Chapter 3.7.11 on page 306
756 Chapter 3.7.11 on page 306
757 Chapter 3.7.11 on page 307
758 Chapter 3.7.11 on page 308
759 Chapter 3.7.11 on page 308
760 Chapter 3.7.11 on page 310
761 Chapter 3.7.11 on page 310
762 Chapter 3.7.11 on page 312
763 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

isupper

Syntax

```
include <cctype> int isupper( int ch );
```

The isupper() function returns non-zero if its argument is an uppercase letter. Otherwise, zero is returned.

Related topics

ISLOWER[764] - TOLOWER[765]

766

isxdigit

Syntax

```
include <cctype> int isxdigit( int ch );
```

The function isxdigit() returns non-zero if its argument is a hexadecimal digit (i.e. A-F, a-f, or 0-9). Otherwise, zero is returned.

Related topics

ISALNUM[767] - ISALPHA[768] - ISCNTRL[769] - ISDIGIT[770] - ISGRAPH[771] - ISPUNCT[772] - ISSPACE[773]

774

764 Chapter 3.7.11 on page 309
765 Chapter 3.7.11 on page 329
766 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
767 Chapter 3.7.11 on page 306
768 Chapter 3.7.11 on page 306
769 Chapter 3.7.11 on page 307
770 Chapter 3.7.11 on page 308
771 Chapter 3.7.11 on page 308
772 Chapter 3.7.11 on page 310
773 Chapter 3.7.11 on page 311
774 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

memchr

Syntax

```
include <cstring> void *memchr( const void *buffer, int ch, size_t count );
```

The memchr() function looks for the first occurrence of *ch* within *count* characters in the array pointed to by *buffer*. The return value points to the location of the first occurrence of *ch*, or **NULL** if ch isn't found. For example:

```
char names[] = "Alan Bob Chris X Dave";
if( memchr(names,'X',strlen(names)) == NULL )
  printf( "Didn't find an X\n" );
else
  printf( "Found an X\n" );
```

Related topics

MEMCMP[775] - MEMCPY[776] - STRSTR[777]

778

memcmp

Syntax

```
include <cstring> int memcmp( const void *buffer1, const void *buffer2, size_t count
);
```

The function memcmp() compares the first *count* characters of *buffer1* and *buffer2*. The return values are as follows:

Return value	Explanation
less than 0	*buffer1* is less than *buffer2*
equal to 0	*buffer1* is equal to *buffer2*

775 Chapter 3.7.11 on page 313
776 Chapter 3.7.11 on page 314
777 Chapter 3.7.11 on page 325
778 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

Return value	Explanation
greater than 0	*buffer1* is greater than *buffer2*

Related topics

MEMCHR[779] - MEMCPY[780] - MEMSET[781] - STRCMP[782]

783

memcpy

Syntax

```
include <cstring> void *memcpy( void *to, const void *from, size_t count );
```

The function memcpy() copies *count* characters from the array *from* to the array *to*. The return value of memcpy() is *to*. The behavior of memcpy() is undefined if *to* and *from* overlap.

Related topics

MEMCHR[784] - MEMCMP[785] - MEMMOVE[786] - MEMSET[787] - STRCPY[788] - STRLEN[789] - STRNCPY[790]

791

779 Chapter 3.7.11 on page 312
780 Chapter 3.7.11 on page 314
781 Chapter 3.7.11 on page 315
782 Chapter 3.7.11 on page 317
783 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
784 Chapter 3.7.11 on page 312
785 Chapter 3.7.11 on page 313
786 Chapter 3.7.11 on page 314
787 Chapter 3.7.11 on page 315
788 Chapter 3.7.11 on page 319
789 Chapter 3.7.11 on page 321
790 Chapter 3.7.11 on page 322
791 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

memmove

Syntax

```
include <cstring> void *memmove( void *to, const void *from, size_t count );
```

The memmove() function is identical to MEMCPY[792](), except that it works even if *to* and *from* overlap.

Related topics

MEMCPY[793] - MEMSET[794]

795

memset

Syntax

```
include <cstring> void* memset( void* buffer, int ch, size_t count );
```

The function memset() copies *ch* into the first *count* characters of *buffer*, and returns *buffer*. memset() is useful for intializing a section of memory to some value. For example, this command:

```
const int ARRAY_LENGTH;
char the_array[ARRAY_LENGTH];
...
// zero out the contents of the_array
memset( the_array, '\0', ARRAY_LENGTH );
```

...is a very efficient way to set all values of the_array to zero.

792 Chapter 3.7.11 on page 314
793 Chapter 3.7.11 on page 314
794 Chapter 3.7.11 on page 315
795 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

The table below compares two different methods for initializing an array of characters: a `for` loop versus memset(). As the size of the data being initialized increases, memset() clearly gets the job done much more quickly:

Input size	Initialized with a `for` loop	Initialized with memset()
1000	0.016	0.017
10000	0.055	0.013
100000	0.443	0.029
1000000	4.337	0.291

Related topics

MEMCMP[796] - MEMCPY[797] - MEMMOVE[798]

799

strcat

Syntax

```
include <cstring> char *strcat( char *str1, const char *str2 );
```

The strcat() function concatenates *str2* onto the end of *str1*, and returns *str1*. For example:

```
printf( "Enter your name: " );
scanf( "%s", name );
title = strcat( name, " the Great" );
printf( "Hello, %s\n", title );  ;
```

Note that strcat() does not perform bounds checking, and thus risks overrunning *str1* or *str2*. For a similar (and safer) function that includes bounds checking, see STRNCAT[800]().

796 Chapter 3.7.11 on page 313
797 Chapter 3.7.11 on page 314
798 Chapter 3.7.11 on page 314
799 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
800 Chapter 3.7.11 on page 321

Related topics

STRCHR[801] - STRCMP[802] - STRCPY[803] - STRNCAT[804]

805

strchr

Syntax

```
include <cstring> char *strchr( const char *str, int ch );
```

The function strchr() returns a pointer to the first occurrence of *ch* in *str*, or **NULL** if *ch* is not found.

Related topics

STRCAT[806] - STRCMP[807] - STRCPY[808] - STRLEN[809] - STRNCAT[810] - STRNCMP[811] - STRNCPY[812] - STRPBRK[813] - STRRCHR[814] -STRSPN[815] - STRSTR[816] - STRTOK[817]

818

801 Chapter 3.7.11 on page 317
802 Chapter 3.7.11 on page 317
803 Chapter 3.7.11 on page 319
804 Chapter 3.7.11 on page 321
805 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
806 Chapter 3.7.11 on page 316
807 Chapter 3.7.11 on page 317
808 Chapter 3.7.11 on page 319
809 Chapter 3.7.11 on page 321
810 Chapter 3.7.11 on page 321
811 Chapter 3.7.11 on page 322
812 Chapter 3.7.11 on page 322
813 Chapter 3.7.11 on page 323
814 Chapter 3.7.11 on page 324
815 Chapter 3.7.11 on page 324
816 Chapter 3.7.11 on page 325
817 Chapter 3.7.11 on page 326
818 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

strcmp

Syntax

```
include <cstring> int strcmp( const char *str1, const char *str2 );
```

The function strcmp() compares *str1* and *str2*, then returns:

Return value	Explanation
less than 0	*str1* is less than *str2*
equal to 0	*str1* is equal to *str2*
greater than 0	*str1* is greater than *str2*

For example:

```
printf( "Enter your name: " );
scanf( "%s", name );
if( strcmp( name, "Mary" ) == 0 ) {
  printf( "Hello, Dr. Mary!\n" );
}
```

Note that if *str1* or *str2* are missing a null-termination character, then strcmp() may not produce valid results. For a similar (and safer) function that includes explicit bounds checking, see strncmp().

Related topics

MEMCMP[819] - STRCAT[820] - STRCHR[821] - STRCOLL[822] - STRCPY[823] - STRLEN[824] - STRNCMP[825] - STRXFRM[826]

827

819 Chapter 3.7.11 on page 313
820 Chapter 3.7.11 on page 316
821 Chapter 3.7.11 on page 317
822 Chapter 3.7.11 on page 318
823 Chapter 3.7.11 on page 319
824 Chapter 3.7.11 on page 321
825 Chapter 3.7.11 on page 322
826 Chapter 3.7.11 on page 328
827 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

strcoll

Syntax

```
include <cstring> int strcoll( const char *str1, const char *str2 );
```

The strcoll() function compares str1 and str2, much like STRCMP[828]. However, strcoll() performs the comparison using the locale specified by the (Standard C Date & Time) SETLOCALE[829]() function.

Related topics

STRCMP[830] - STRXFRM[831]
(Standard C Date & Time) SETLOCALE[832]

833

strcpy

Syntax

```
include <cstring> char *strcpy( char *to, const char *from );
```

The strcpy() function copies characters in the string **from***to the string* **to***, including the null termination. The return value is* **to***.*

Note that strcpy() does not perform bounds checking, and thus risks overrunning *from* or *to*. For a similar (and safer) function that includes bounds checking, see STRNCPY[834]().

Related topics

828 Chapter 3.7.11 on page 317
829 Chapter 3.7.11 on page 350
830 Chapter 3.7.11 on page 317
831 Chapter 3.7.11 on page 328
832 Chapter 3.7.11 on page 350
833 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
834 Chapter 3.7.11 on page 322

MEMCPY[835] - STRCAT[836] - STRCHR[837] - STRCMP[838] - STRNCMP[839] - STRNCPY[840]

841

strcspn

Syntax

```
include <cstring> size_t strcspn( const char *str1, const char *str2 );
```

The function strcspn() returns the index of the first character in *str1* that matches any of the characters in *str2*.

Related topics

STRPBRK[842] - STRRCHR[843] - STRSTR[844] - STRTOK[845]

846

strerror

Syntax

```
include <cstring> char *strerror( int num );
```

835 Chapter 3.7.11 on page 314
836 Chapter 3.7.11 on page 316
837 Chapter 3.7.11 on page 317
838 Chapter 3.7.11 on page 317
839 Chapter 3.7.11 on page 322
840 Chapter 3.7.11 on page 322
841 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
842 Chapter 3.7.11 on page 323
843 Chapter 3.7.11 on page 324
844 Chapter 3.7.11 on page 325
845 Chapter 3.7.11 on page 326
846 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

The function strerror() returns an implementation defined string corresponding to *num*. If an error occurred, the error is located within the global variable **errno**.

Related topics

PERROR[847]

848

strlen

Syntax

```
include <cstring> size_t strlen( char *str );
```

The strlen() function returns the length of *str* (determined by the number of characters before null termination).

Related topics

MEMCPY[849] - STRCHR[850] - STRCMP[851] - STRNCMP[852]

853

strncat

Syntax

```
include <cstring> char *strncat( char *str1, const char *str2, size_t count );
```

The function strncat() concatenates at most *count* characters of *str2* onto *str1*, adding a null termination. The resulting string is returned.

847 Chapter 3.7.11 on page 289
848 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
849 Chapter 3.7.11 on page 314
850 Chapter 3.7.11 on page 317
851 Chapter 3.7.11 on page 317
852 Chapter 3.7.11 on page 322
853 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

Related topics

STRCAT[854] - STRCHR[855] - STRNCMP[856] - STRNCPY[857]

858

strncmp

Syntax

```
include <cstring> int strncmp( const char *str1, const char *str2, size_t count );
```

The strncmp() function compares at most count characters of *str1* and *str2*. The return value is as follows:

Return value	Explanation
less than 0	*str1* is less than *str2*
equal to 0	*str1* is equal to *str2*
greater than 0	*str1* is greater than *str2*

If there are less than count characters in either string, then the comparison will stop after the first null termination is encountered.

Related topics

STRCHR[859] - STRCMP[860] - STRCPY[861] - STRLEN[862] - STRNCAT[863] - STRNCPY[864]

865

---

854 Chapter 3.7.11 on page 316
855 Chapter 3.7.11 on page 317
856 Chapter 3.7.11 on page 322
857 Chapter 3.7.11 on page 322
858 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
859 Chapter 3.7.11 on page 317
860 Chapter 3.7.11 on page 317
861 Chapter 3.7.11 on page 319
862 Chapter 3.7.11 on page 321
863 Chapter 3.7.11 on page 321
864 Chapter 3.7.11 on page 322
865 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

strncpy

Syntax

```
include <cstring> char *strncpy( char *to, const char *from, size_t count );
```

The strncpy() function copies at most *count* characters of *from* to the string *to*. Only if *from* has less than *count* characters, is the remainder padded with '\0' characters. The *return* value is the resulting string.

Note:
Using strings not padded with the '\0' character can create security vulnerabilities.

Related topics

MEMCPY[866] - STRCHR[867] - STRCPY[868] - STRNCAT[869] - STRNCMP[870]

871

strpbrk

Syntax

```
include <cstring> char * strpbrk( const char *str, const char *ch );
```

The function strchr() returns a pointer to the first occurrence of any character within *ch* in *str*, or **NULL** if no characters were not found.

Related topics

866 Chapter 3.7.11 on page 314
867 Chapter 3.7.11 on page 317
868 Chapter 3.7.11 on page 319
869 Chapter 3.7.11 on page 321
870 Chapter 3.7.11 on page 322
871 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

STRCHR[872] - STRRCHR[873] - STRSTR[874]

875

strrchr

Syntax

```
include <cstring> char *strrchr( const char *str, int ch );
```

The function strrchr() returns a pointer to the last occurrence of *ch* in *str*, or **NULL** if no match is found.

Related topics

STRCHR[876] - STRCSPN[877] - STRPBRK[878] - STRSPN[879] - STRSTR[880] - STRTOK[881]

882

strspn

Syntax

```
include <cstring> size_t strspn( const char *str1, const char *str2 );
```

872 Chapter 3.7.11 on page 324
873 Chapter 3.7.11 on page 324
874 Chapter 3.7.11 on page 325
875 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
876 Chapter 3.7.11 on page 317
877 Chapter 3.7.11 on page 320
878 Chapter 3.7.11 on page 323
879 Chapter 3.7.11 on page 324
880 Chapter 3.7.11 on page 325
881 Chapter 3.7.11 on page 326
882 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

The strspn() function returns the index of the first character in *str1* that doesn't match any character in *str2*.

Related topics

STRCHR[883] - STRPBRK[884] - STRRCHR[885] - STRSTR[886] - STRTOK[887]

888

strstr

Syntax

```
include <cstring> char *strstr( const char *str1, const char *str2 );
```

The function strstr() returns a pointer to the first occurrence of *str2* in *str1*, or **NULL** if no match is found. If the length of *str2* is zero, then strstr() will simply return *str1*.

For example, the following code checks for the existence of one string within another string:

```
char* str1 = "this is a string of characters";
char* str2 = "a string";
char* result = strstr( str1, str2 );
if( result == NULL ) printf( "Could not find '%s' in '%s'\n", str2, str1 );
  else printf( "Found a substring: '%s'\n", result );
```

When run, the above code displays this output:

```
Found a substring: 'a string of characters'
```

Related topics

883 Chapter 3.7.11 on page 317
884 Chapter 3.7.11 on page 323
885 Chapter 3.7.11 on page 324
886 Chapter 3.7.11 on page 325
887 Chapter 3.7.11 on page 326
888 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

MEMCHR[889] - STRCHR[890] - STRCSPN[891] - STRPBRK[892] - STRRCHR[893] - STRSPN[894] - STRTOK[895]

896

strtod

Syntax

```
include <cstdlib> double strtod( const char *start, char **end );
```

The function strtod() returns whatever it encounters first in start as a double. end is set to point at whatever is left in start after that double. If overflow occurs, strtod() returns either **HUGE_VAL** or **-HUGE_VAL**.

```
x = atof( "42.0is_the_answer" );
```

results in x being set to 42.0.

Related topics

ATOF[897]

898

889 Chapter 3.7.11 on page 312
890 Chapter 3.7.11 on page 317
891 Chapter 3.7.11 on page 320
892 Chapter 3.7.11 on page 323
893 Chapter 3.7.11 on page 324
894 Chapter 3.7.11 on page 324
895 Chapter 3.7.11 on page 326
896 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
897 Chapter 3.7.11 on page 304
898 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

strtok

Syntax

```
include <cstring> char *strtok( char *str1, const char *str2 );
```

The strtok() function returns a pointer to the next "token" in *str1*, where *str2* contains the delimiters that determine the token. strtok() returns **NULL** if no token is found. In order to convert a string to tokens, the first call to strtok() should have *str1* point to the string to be tokenized. All calls after this should have *str1* be **NULL**.

For example:

```
char str[] = "now # is the time for all # good men to come to the # aid of their
 country";
char delims[] = "#";
char *result = NULL;
result = strtok( str, delims );
while( result != NULL ) {
  printf( "result is \"%s\"\n", result );
  result = strtok( NULL, delims );
}
```

The above code will display the following output:

```
   result is "now "
   result is " is the time for all "
   result is " good men to come to the "
   result is " aid of their country"
```

Related topics

STRCHR[899] - STRCSPN[900] - STRPBRK[901] - STRRCHR[902] - STRSPN[903] - STRSTR[904]

905

899 Chapter 3.7.11 on page 317
900 Chapter 3.7.11 on page 320
901 Chapter 3.7.11 on page 323
902 Chapter 3.7.11 on page 324
903 Chapter 3.7.11 on page 324
904 Chapter 3.7.11 on page 325
905 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

strtol

Syntax

```
include <cstdlib> long strtol( const char *start, char **end, int base );
```

The strtol() function returns whatever it encounters first in *start* as a long, doing the conversion to *base* if necessary. *end* is set to point to whatever is left in *start* after the long. If the result can not be represented by a long, then strtol() returns either **LONG_MAX** or **LONG_MIN**. Zero is returned upon error.

Related topics

ATOL[906] - STRTOUL[907]

908

strtoul

Syntax

```
include <cstdlib> unsigned long strtoul( const char *start, char **end, int base );
```

The function strtoul() behaves exactly like STRTOL[909](), except that it returns an `unsigned` long rather than a mere long.

Related topics

STRTOL[910]

911

906 Chapter 3.7.11 on page 305
907 Chapter 3.7.11 on page 328
908 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
909 Chapter 3.7.11 on page 327
910 Chapter 3.7.11 on page 327
911 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

strxfrm

Syntax

```
include <cstring> size_t strxfrm( char *str1, const char *str2, size_t num );
```

The strxfrm() function manipulates the first *num* characters of *str2* and stores them in *str1*. The result is such that if a STRCOLL[912]() is performed on *str1* and the old *str2*, you will get the same result as with a STRCMP[913]().

Related topics

STRCMP[914] - STRCOLL[915]

916

tolower

Syntax

```
include <cctype> int tolower( int ch );
```

The function tolower() returns the lowercase version of the character *ch*.

Related topics

ISUPPER[917] - TOUPPER[918]

919

912 Chapter 3.7.11 on page 318
913 Chapter 3.7.11 on page 317
914 Chapter 3.7.11 on page 317
915 Chapter 3.7.11 on page 318
916 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
917 Chapter 3.7.11 on page 311
918 Chapter 3.7.11 on page 329
919 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

toupper

Syntax

```
include <cctype> int toupper( int ch );
```

The toupper() function returns the uppercase version of the character *ch*.

Related topics

TOLOWER[920]

921

Standard C Math

This section will cover the Math elements of the C Standard Library.

abs

Syntax

```
include <cstdlib> int abs( int num );
```

The abs() function returns the absolute value of *num*. For example:

```
int magic_number = 10;
cout << "Enter a guess: ";
cin >> x;
cout << "Your guess was " << abs( magic_number - x ) << " away from the magic
 number." << endl;
```

Related topics

FABS[922] - LABS[923]

920 Chapter 3.7.11 on page 329
921 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
922 Chapter 3.7.11 on page 336
923 Chapter 3.7.11 on page 338

924

acos

Syntax

```
include <cmath> double acos( double arg );
```

The acos() function returns the arc cosine of *arg*, which will be in the range [0, pi]. *arg* should be between -1 and 1. If *arg* is outside this range, acos() returns NAN and raises a floating-point exception.

Related topics

ASIN[925] - ATAN[926] - ATAN2[927] - COS[928] - COSH[929] - SIN[930] - SINH[931] - TAN[932] - TANH[933]

934

asin

Syntax

```
include <cmath> double asin( double arg );
```

924 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
925 Chapter 3.7.11 on page 331
926 Chapter 3.7.11 on page 332
927 Chapter 3.7.11 on page 333
928 Chapter 3.7.11 on page 334
929 Chapter 3.7.11 on page 334
930 Chapter 3.7.11 on page 342
931 Chapter 3.7.11 on page 343
932 Chapter 3.7.11 on page 344
933 Chapter 3.7.11 on page 345
934 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

The asin() function returns the arc sine of *arg*, which will be in the range [-pi/2, +pi/2]. *arg* should be between -1 and 1. If *arg* is outside this range, asin() returns NAN and raises a floating-point exception.

Related topics

ACOS[935] - ATAN[936] - ATAN2[937] - COS[938] - COSH[939] - SIN[940] - SINH[941] - TAN[942] - TANH[943]

944

atan

Syntax

```
include <cmath> double atan( double arg );
```

The function atan() returns the arc tangent of *arg*, which will be in the range [-pi/2, +pi/2].

Related topics

ACOS[945] - ASIN[946] - ATAN2[947] - COS[948] - COSH[949] - SIN[950] - SINH[951] - TAN[952] - TANH[953]

---

935 Chapter 3.7.11 on page 331
936 Chapter 3.7.11 on page 332
937 Chapter 3.7.11 on page 333
938 Chapter 3.7.11 on page 334
939 Chapter 3.7.11 on page 334
940 Chapter 3.7.11 on page 342
941 Chapter 3.7.11 on page 343
942 Chapter 3.7.11 on page 344
943 Chapter 3.7.11 on page 345
944 http://en.wikibooks.org/wiki/Category%3AC%2B%2B%20Programming
945 Chapter 3.7.11 on page 331
946 Chapter 3.7.11 on page 331
947 Chapter 3.7.11 on page 333
948 Chapter 3.7.11 on page 334
949 Chapter 3.7.11 on page 334
950 Chapter 3.7.11 on page 342
951 Chapter 3.7.11 on page 343
952 Chapter 3.7.11 on page 344
953 Chapter 3.7.11 on page 345

954

atan2

Syntax

```
include <cmath> double atan2( double y, double x );
```

The atan2() function computes the arc tangent of y/x, using the signs of the arguments to compute the quadrant of the return value.

Related topics

ACOS[955] - ASIN[956] - ATAN[957] - COS[958] - COSH[959] - SIN[960] - SINH[961] - TAN[962] - TANH[963]

964

ceil

Syntax

```
include <cmath> double ceil( double num );
```

The ceil() function returns the smallest integer no less than num. For example:

954 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
955 Chapter 3.7.11 on page 331
956 Chapter 3.7.11 on page 331
957 Chapter 3.7.11 on page 332
958 Chapter 3.7.11 on page 334
959 Chapter 3.7.11 on page 334
960 Chapter 3.7.11 on page 342
961 Chapter 3.7.11 on page 343
962 Chapter 3.7.11 on page 344
963 Chapter 3.7.11 on page 345
964 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

```
y = 6.04;
x = ceil( y );
```

would set x to 7.0.

Related topics

FLOOR[965] - FMOD[966]

967

cos

Syntax

```
include <cmath> float cos( float arg ); double cos( double arg ); long double cos(
long double arg );
```

The cos() function returns the cosine of *arg*, where *arg* is expressed in radians. The return value of cos() is in the range [-1,1]. If *arg* is infinite, cos() will return NAN and raise a floating-point exception.

Related topics

ACOS[968] - ASIN[969] - ATAN[970] - ATAN2[971] - COSH[972] - SIN[973] - SINH[974] - TAN[975] - TANH[976]

977

965 Chapter 3.7.11 on page 337
966 Chapter 3.7.11 on page 337
967 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
968 Chapter 3.7.11 on page 331
969 Chapter 3.7.11 on page 331
970 Chapter 3.7.11 on page 332
971 Chapter 3.7.11 on page 333
972 Chapter 3.7.11 on page 334
973 Chapter 3.7.11 on page 342
974 Chapter 3.7.11 on page 343
975 Chapter 3.7.11 on page 344
976 Chapter 3.7.11 on page 345
977 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

cosh

Syntax

```
include <cmath> float cosh( float arg ); double cosh(
double arg ); long double cosh( long double arg );
```

The function cosh() returns the hyperbolic cosine of *arg*.

Related topics

ACOS[978] - ASIN[979] - ATAN[980] - ATAN2[981] - COS[982] - SIN[983] - SINH[984] - TAN[985] - TANH[986]

987

div

Syntax

```
include <cstdlib> div_t div( int numerator, int denominator );
```

The function div() returns the quotient and remainder of the operation *numerator* / *denominator*. The **div_t** structure is defined in cstdlib, and has at least:

```
int quot;   // The quotient
int rem;    // The remainder
```

978 Chapter 3.7.11 on page 331
979 Chapter 3.7.11 on page 331
980 Chapter 3.7.11 on page 332
981 Chapter 3.7.11 on page 333
982 Chapter 3.7.11 on page 334
983 Chapter 3.7.11 on page 342
984 Chapter 3.7.11 on page 343
985 Chapter 3.7.11 on page 344
986 Chapter 3.7.11 on page 345
987 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

For example, the following code displays the quotient and remainder of x/y:

```
div_t temp;
temp = div( x, y );
printf( "%d divided by %d yields %d with a remainder of %d\n",
  x, y, temp.quot, temp.rem );
```

Related topics

LDIV[988]

989

exp

Syntax

```
include <cmath> double exp( double arg );
```

The exp() function returns e (2.7182818) raised to the *arg*th power.

Related topics

LOG[990] - POW[991] - SQRT[992]

993

fabs

Syntax

```
include <cmath> double fabs( double arg );
```

The function fabs() returns the absolute value of *arg*.

988 Chapter 3.7.11 on page 339
989 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
990 Chapter 3.7.11 on page 340
991 Chapter 3.7.11 on page 342
992 Chapter 3.7.11 on page 343
993 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

Related topics

ABS[994] - FMOD[995] - LABS[996]

997

floor

Syntax

```
include <cmath> double floor( double arg );
```

The function floor() returns the largest integer value not greater than arg.

```
// Example for positive numbers
y = 6.04;
x = floor( y );
```

would result in x being set to 6 (double 6.0).

```
// Example for negative numbers
y = -6.04;
x = floor( y );
```

would result in x being set to -7 (double -7.0).

Related topics

CEIL[998] - FMOD[999]

1000

994 Chapter 3.7.11 on page 330
995 Chapter 3.7.11 on page 337
996 Chapter 3.7.11 on page 338
997 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
998 Chapter 3.7.11 on page 333
999 Chapter 3.7.11 on page 337
1000 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

fmod

Syntax

```
include <cmath> double fmod( double x, double y );
```

The fmod() function returns the remainder of x/y.

Related topics

CEIL[1001] - FABS[1002] - FLOOR[1003]

1004

frexp

Syntax

```
include <cmath> double frexp( double num, int* exp );
```

The function frexp() is used to decompose *num* into two parts: a mantissa between 0.5 and 1 (returned by the function) and an exponent returned as *exp*. Scientific notation works like this:

```
num = mantissa * (2 ^ exp)
```

Related topics

LDEXP[1005] - MODF[1006]

1007

1001 Chapter 3.7.11 on page 333
1002 Chapter 3.7.11 on page 336
1003 Chapter 3.7.11 on page 337
1004 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
1005 Chapter 3.7.11 on page 339
1006 Chapter 3.7.11 on page 341
1007 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

labs

Syntax

```
include <cstdlib> long labs( long num );
```

The function labs() returns the absolute value of *num*.

Related topics

ABS[1008] - FABS[1009]

1010

ldexp

Syntax

```
include <cmath> double ldexp( double num, int exp );
```

The ldexp() function returns *num* * (2 ^ *exp*). And get this: if an overflow occurs, **HUGE_VAL** is returned.

Related topics

FREXP[1011] - MODF[1012]

1013

1008 Chapter 3.7.11 on page 330
1009 Chapter 3.7.11 on page 336
1010 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
1011 Chapter 3.7.11 on page 338
1012 Chapter 3.7.11 on page 341
1013 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

ldiv

Syntax

```
include <cstdlib> ldiv_t ldiv( long numerator, long denominator );
```

Testing: **adiv_t**, **div_t**, **ldiv_t**.

The ldiv() function returns the quotient and remainder of the operation *numerator / denominator*. The **ldiv_t** structure is defined in cstdlib and has at least:

```
long quot;  // the quotient
long rem;   // the remainder
```

Related topics

DIV[1014]

1015

log

Syntax

```
include <cmath> double log( double num );
```

The function log() returns the natural (base e) logarithm of *num*. There's a domain error if *num* is negative, a range error if *num* is zero.

In order to calculate the logarithm of x to an arbitrary base b, you can use:

```
double answer = log(x) / log(b);
```

Related topics

1014 Chapter 3.7.11 on page 335
1015 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

EXP[1016] - LOG10[1017] - POW[1018] - SQRT[1019]

1020

log10

Syntax

```
include <cmath> double log10( double num );
```

The `log10()` function returns the base 10 (or common) logarithm for `num`. There will be a domain error if `num` is negative and a range error if `num` is zero.

Related topics

LOG[1021]

1022

modf

Syntax

```
include <cmath> double modf( double num, double *i );
```

The function modf() splits *num* into its integer and fraction parts. It returns the fractional part and loads the integer part into *i*.

Related topics

1016 Chapter 3.7.11 on page 336
1017 Chapter 3.7.11 on page 341
1018 Chapter 3.7.11 on page 342
1019 Chapter 3.7.11 on page 343
1020 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
1021 Chapter 3.7.11 on page 340
1022 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

FREXP[1023] - LDEXP[1024]

1025

pow

Syntax

```
include <cmath> double pow( double base, double exp );
```

The pow() function returns *base* raised to the *exp*th power. There's a domain error if base is zero and *exp* is less than or equal to zero. There's also a domain error if base is negative and *exp* is not an integer. There's a range error if an overflow occurs.

Related topics

EXP[1026] - LOG[1027] - SQRT[1028]

1029

sin

Syntax

```
include <cmath> double sin( double arg );
```

The function sin() returns the sine of *arg*, where *arg* is given in radians. The return value of sin() will be in the range [-1,1]. If arg is infinite, sin() will return NAN and raise a floating-point exception.

1023 Chapter 3.7.11 on page 338
1024 Chapter 3.7.11 on page 339
1025 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
1026 Chapter 3.7.11 on page 336
1027 Chapter 3.7.11 on page 340
1028 Chapter 3.7.11 on page 343
1029 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

Related topics

ACOS[1030] - ASIN[1031] - ATAN[1032] - ATAN2[1033] - COS[1034] - COSH[1035] - SINH[1036] - TAN[1037] - TANH[1038]

1039

sinh

Syntax

```
include <cmath> double sinh( double arg );
```

The function sinh() returns the hyperbolic sine of *arg*.

Related topics

ACOS[1040] - ASIN[1041] - ATAN[1042] - ATAN2[1043] - COS[1044] - COSH[1045] - SIN[1046] - TAN[1047] - TANH[1048]

1049

1030 Chapter 3.7.11 on page 331
1031 Chapter 3.7.11 on page 331
1032 Chapter 3.7.11 on page 332
1033 Chapter 3.7.11 on page 333
1034 Chapter 3.7.11 on page 334
1035 Chapter 3.7.11 on page 334
1036 Chapter 3.7.11 on page 343
1037 Chapter 3.7.11 on page 344
1038 Chapter 3.7.11 on page 345
1039 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
1040 Chapter 3.7.11 on page 331
1041 Chapter 3.7.11 on page 331
1042 Chapter 3.7.11 on page 332
1043 Chapter 3.7.11 on page 333
1044 Chapter 3.7.11 on page 334
1045 Chapter 3.7.11 on page 334
1046 Chapter 3.7.11 on page 342
1047 Chapter 3.7.11 on page 344
1048 Chapter 3.7.11 on page 345
1049 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

sqrt

Syntax

```
include <cmath> double sqrt( double num );
```

The sqrt() function returns the square root of *num*. If *num* is negative, a domain error occurs.

Related topics

EXP[1050] - LOG[1051] - POW[1052]

1053

tan

Syntax

```
include <cmath> double tan( double arg );
```

The tan() function returns the tangent of *arg*, where *arg* is given in radians. If *arg* is infinite, tan() will return NAN and raise a floating-point exception.

Related topics

ACOS[1054] - ASIN[1055] - ATAN[1056] - ATAN2[1057] - COS[1058] - COSH[1059] - SIN[1060] - SINH[1061] - TANH[1062]

1050 Chapter 3.7.11 on page 336
1051 Chapter 3.7.11 on page 340
1052 Chapter 3.7.11 on page 342
1053 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
1054 Chapter 3.7.11 on page 331
1055 Chapter 3.7.11 on page 331
1056 Chapter 3.7.11 on page 332
1057 Chapter 3.7.11 on page 333
1058 Chapter 3.7.11 on page 334
1059 Chapter 3.7.11 on page 334
1060 Chapter 3.7.11 on page 342
1061 Chapter 3.7.11 on page 343
1062 Chapter 3.7.11 on page 345

1063

tanh

Syntax

```
include <cmath> double tanh( double arg );
```

```
/*example*/
#include <stdio.h>
#include <math.h>
int main (){
        double c, p;
        c = log(2.0);
        p = tanh (c);
        printf ("The hyperbolic tangent of %lf is %lf.\n", c, p );
return 0;
}
```

The function tanh() returns the hyperbolic tangent of *arg*.

Related topics

ACOS[1064] - ASIN[1065] - ATAN[1066] - ATAN2[1067] - COS[1068] - COSH[1069] - SIN[1070] - SINH[1071] - TAN[1072]

1073

Standard C Time & Date

This section will cover the Time and Date elements of the C Standard Library.

1063 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
1064 Chapter 3.7.11 on page 331
1065 Chapter 3.7.11 on page 331
1066 Chapter 3.7.11 on page 332
1067 Chapter 3.7.11 on page 333
1068 Chapter 3.7.11 on page 334
1069 Chapter 3.7.11 on page 334
1070 Chapter 3.7.11 on page 342
1071 Chapter 3.7.11 on page 343
1072 Chapter 3.7.11 on page 344
1073 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

asctime

Syntax

```
include <ctime> char *asctime( const struct tm *ptr );
```

The function asctime() converts the time in the struct 'ptr' to a character string of the following format:

```
day month date hours:minutes:seconds year
```

An example:

```
Mon Jun 26 12:03:53 2000
```

Related topics

CLOCK[1074] - CTIME[1075] - DIFFTIME[1076] - GMTIME[1077] - LOCALTIME[1078] - MKTIME[1079] - TIME[1080]

1081

clock

Syntax

```
include <ctime> clock_t clock( void );
```

1074 Chapter 3.7.11 on page 346
1075 Chapter 3.7.11 on page 347
1076 Chapter 3.7.11 on page 348
1077 Chapter 3.7.11 on page 348
1078 Chapter 3.7.11 on page 349
1079 Chapter 3.7.11 on page 349
1080 Chapter 3.7.11 on page 352
1081 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

The clock() function returns the processor time since the program started, or -1 if that information is unavailable. To convert the return value to seconds, divide it by **CLOCKS_PER_SEC**.

Note:
If your compiler and library is POSIX compliant, then **CLOCKS_PER_SEC** is always defined as 1000000.

Related topics

ASCTIME[1082] - CTIME[1083] - TIME[1084]

1085

ctime

Syntax

```
include <ctime> char *ctime( const time_t *time );
```

The ctime() function converts the calendar time *time* to local time of the format:

```
day month date hours:minutes:seconds year
```

using ctime() is equivalent to

```
asctime( localtime( tp ) );
```

Related topics

1082 Chapter 3.7.11 on page 345
1083 Chapter 3.7.11 on page 347
1084 Chapter 3.7.11 on page 352
1085 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

ASCTIME[1086] - CLOCK[1087] - GMTIME[1088] - LOCALTIME[1089] - MKTIME[1090] - TIME[1091]

1092

difftime

Syntax

```
include <ctime> double difftime( time_t time2, time_t time1 );
```

The function difftime() returns *time2 - time1*, in seconds.

Related topics

ASCTIME[1093] - GMTIME[1094] - LOCALTIME[1095] - TIME[1096]

1097

gmtime

Syntax

```
include <ctime> struct tm *gmtime( const time_t *time );
```

1086 Chapter 3.7.11 on page 345
1087 Chapter 3.7.11 on page 346
1088 Chapter 3.7.11 on page 348
1089 Chapter 3.7.11 on page 349
1090 Chapter 3.7.11 on page 349
1091 Chapter 3.7.11 on page 352
1092 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
1093 Chapter 3.7.11 on page 345
1094 Chapter 3.7.11 on page 348
1095 Chapter 3.7.11 on page 349
1096 Chapter 3.7.11 on page 352
1097 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

The gmtime() function returns the given time in Coordinated Universal Time (usually Greenwich mean time), unless it's not supported by the system, in which case **NULL** is returned. Watch out for the STATIC RETURN[1098].

Related topics

ASCTIME[1099] - CTIME[1100] - DIFFTIME[1101] - LOCALTIME[1102] - MKTIME[1103] - STRFTIME[1104] - TIME[1105]

1106

localtime

Syntax

```
include <ctime> struct tm *localtime( const time_t *time );
```

The function localtime() converts calendar time *time* into local time. Watch out for the STATIC RETURN[1107].

Related topics

ASCTIME[1108] - CTIME[1109] - DIFFTIME[1110] - GMTIME[1111] - STRFTIME[1112] - TIME[1113]

1114

1098 Chapter 3.7.4 on page 246
1099 Chapter 3.7.11 on page 345
1100 Chapter 3.7.11 on page 347
1101 Chapter 3.7.11 on page 348
1102 Chapter 3.7.11 on page 349
1103 Chapter 3.7.11 on page 349
1104 Chapter 3.7.11 on page 351
1105 Chapter 3.7.11 on page 352
1106 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
1107 Chapter 3.7.4 on page 246
1108 Chapter 3.7.11 on page 345
1109 Chapter 3.7.11 on page 347
1110 Chapter 3.7.11 on page 348
1111 Chapter 3.7.11 on page 348
1112 Chapter 3.7.11 on page 351
1113 Chapter 3.7.11 on page 352
1114 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

mktime

Syntax

```
include <ctime> time_t mktime( struct tm *time );
```

The mktime() function converts the local time in *time* to calendar time, and returns it. If there is an error, -1 is returned.

Related topics

ASCTIME[1115] - CTIME[1116] - GMTIME[1117] - TIME[1118]

1119

setlocale

Syntax

```
include <clocale> char *setlocale( int category, const char * locale );
```

The setlocale() function is used to set and retrieve the current locale. If *locale* is **NULL**, the current locale is returned. Otherwise, *locale* is used to set the locale for the given *category*.

category can have the following values:

Value	Description
LC_ALL	All of the locale
LC_TIME	Date and time formatting
LC_NUMERIC	Number formatting

1115 Chapter 3.7.11 on page 345
1116 Chapter 3.7.11 on page 347
1117 Chapter 3.7.11 on page 348
1118 Chapter 3.7.11 on page 352
1119 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

Value	Description
LC_COLLATE	String collation and regular expression matching
LC_CTYPE	Regular expression matching, conversion, case-sensitive comparison, wide character functions, and character classification.
LC_MONETARY	For monetary formatting
LC_MESSAGES	For natural language messages

Related topics

(Standard C String & Character) STRCOLL[1120]

1121

strftime

Syntax

```
include <ctime> size_t strftime( char *str, size_t maxsize, const char *fmt,
struct tm *time );
```

The function strftime() formats date and time information from time to a format specified by *fmt*, then stores the result in *str* (up to *maxsize* characters). Certain codes may be used in *fmt* to specify different types of *time*:

Code	Meaning
%a	abbreviated weekday name (e.g. Fri)
%A	full weekday name (e.g. Friday)
%b	abbreviated month name (e.g. Oct)
%B	full month name (e.g. October)
%c	the standard date and time string
%d	day of the month, as a number (1-31)

1120 Chapter 3.7.11 on page 318
1121 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

Code	Meaning
%H	hour, 24 hour format (0-23)
%I	hour, 12 hour format (1-12)
%j	day of the year, as a number (1-366)
%m	month as a number (1-12).
%M	minute as a number (0-59)
%p	locale's equivalent of AM or PM
%S	second as a number (0-59)
%U	week of the year, (0-53), where week 1 has the first Sunday
%w	weekday as a decimal (0-6), where Sunday is 0
%W	week of the year, (0-53), where week 1 has the first Monday
%x	standard date string
%X	standard time string
%y	year in decimal, without the century (0-99)
%Y	year in decimal, with the century
%Z	time zone name
%%	a percent sign

Note:
Some versions of Microsoft Visual C++ may use values that range from 0-11 to describe %m (month as a number).

Related topics

GMTIME[1122] - LOCALTIME[1123] - TIME[1124]

[1125]

1122 Chapter 3.7.11 on page 348
1123 Chapter 3.7.11 on page 349
1124 Chapter 3.7.11 on page 352
1125 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

time

Syntax

```
include <ctime> time_t time( time_t *time );
```

The function time() returns the current time, or -1 if there is an error. If the argument *time* is given, then the current time is stored in *time*.

Related topics

ASCTIME[1126] - CLOCK[1127] - CTIME[1128] - DIFFTIME[1129] - GMTIME[1130] - LOCALTIME[1131] - MKTIME[1132] - STRFTIME[1133]
(Other Standard C functions) SRAND[1134]

1135

Standard C Memory Management

This section will cover memory management elements from the Standard C Library.

Note:
It is recommended to use the new and delete operators instead of these functions, as they provide additional control over the creation of objects.

1126 Chapter 3.7.11 on page 345
1127 Chapter 3.7.11 on page 346
1128 Chapter 3.7.11 on page 347
1129 Chapter 3.7.11 on page 348
1130 Chapter 3.7.11 on page 348
1131 Chapter 3.7.11 on page 349
1132 Chapter 3.7.11 on page 349
1133 Chapter 3.7.11 on page 351
1134 Chapter 3.7.11 on page 364
1135 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

calloc

Syntax

```
include <cstdlib> void *calloc( size_t num, size_t size);
```

The function calloc() allocates a block of memory that can store *num* objects of size *size*. In addition, the block of memory allocated is set to all zeros.

If the operation fails, calloc() returns **NULL**.

Related topics

FREE[1136] - MALLOC[1137] - REALLOC[1138]

1139

free

Syntax

```
include <cstdlib> void free( void *p);
```

The function free() releases a previously allocated block from a call to calloc, malloc, or realloc.

Related topics

CALLOC[1140] - MALLOC[1141] - REALLOC[1142]

1143

1136 Chapter 3.7.11 on page 354
1137 Chapter 3.7.11 on page 354
1138 Chapter 3.7.11 on page 355
1139 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
1140 Chapter 3.7.11 on page 353
1141 Chapter 3.7.11 on page 354
1142 Chapter 3.7.11 on page 355
1143 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

malloc

Syntax

```
include <cstdlib> void *malloc( size_t s );
```

The function malloc() allocates a block of memory of size *s*. The memory remains uninitialized.

If the operation fails, malloc() returns **NULL**.

Related topics

CALLOC[1144] - FREE[1145] - REALLOC[1146]

1147

realloc

Syntax

```
include <cstdlib> void *realloc( void *p, size_t s);
```

The function realloc() resizes a block created by malloc() or calloc(), and returns a pointer to the new memory region.

If the resize operation fails, realloc() returns **NULL** and leaves the old memory region intact.

Note:
realloc() does not have a corresponding operator in C++ - however, this is not required since the standard template library already provides the necessary memory management for most usages.

1144 Chapter 3.7.11 on page 353
1145 Chapter 3.7.11 on page 354
1146 Chapter 3.7.11 on page 355
1147 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

Related topics

CALLOC[1148] - FREE[1149] - MALLOC[1150]

1151

1152

Other Standard C functions

This section will cover several functions that are outside of the previous niches but are nevertheless part of the C Standard Library.

abort

Syntax

```
include <cstdlib> void abort( void );
```

The function abort() terminates the current program. Depending on the implementation, the return from the function can indicate a canceled (e.g. you used the signal() function to catch **SIGABRT**) or failed abort.

SIGABRT is sent by the process to itself when it calls the abort libc function, defined in cstdlib. The **SIGABRT** signal can be caught, but it cannot be blocked; if the signal handler returns then all open streams are closed and flushed and the program terminates (dumping core if appropriate). This means that the abort call never returns. Because of this characteristic, it is often used to signal fatal conditions in support libraries, situations where the current operation cannot be completed but the main program can perform cleanup before exiting. It is also used if an assertion fails.

Related topics

1148 Chapter 3.7.11 on page 353
1149 Chapter 3.7.11 on page 354
1150 Chapter 3.7.11 on page 354
1151 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
1152 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

ASSERT[1153] - ATEXIT[1154] - EXIT[1155]

1156

assert

Syntax

```
include <cassert> assert( exp );
```

The assert() macro is used to test for errors. If exp evaluates to zero, assert() writes information to stderr and exits the program. If the macro NDEBUG is defined, the assert() macros will be ignored.

Related topics

ABORT[1157]

1158

atexit

Syntax

```
include <cstdlib> int atexit( void (*func)(void) );
```

The function atexit() causes the function pointed to by func to be called when the program terminates. You can make multiple calls to atexit() (at least 32, depending on your compiler) and those functions will be called in reverse order of their establishment. The return value of atexit() is zero upon success, and non-zero on failure.

1153 Chapter 3.7.11 on page 357
1154 Chapter 3.7.11 on page 357
1155 Chapter 3.7.11 on page 358
1156 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
1157 Chapter 3.7.11 on page 356
1158 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

Related topics

ABORT[1159] - EXIT[1160]

1161

bsearch

Syntax

```
include <cstdlib> void* bsearch( const void *key, const void *base,
size_t num, size_t size, int (*compare)(const void *, const void *));
```

The function bsearch() performs a search within a sorted array, returning a pointer to the element in question or **NULL**.

**key* refers to an object that matches an item searched within **base*. This array contains *num* elements, each of size *size*.

The *compare* function accepts two pointers to the object within the array - which need to first be cast to the object type being examined. The function returns -1 if the first parameter should be before the second, 1 if the first parameter is after, or 0 if the object matches.

Related topics

QSORT[1162]

1163

1159 Chapter 3.7.11 on page 356
1160 Chapter 3.7.11 on page 358
1161 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
1162 Chapter 3.7.11 on page 360
1163 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

exit

Syntax

```
include <cstdlib> void exit( int exit_code );
```

The exit() function stops the program. exit_code is passed on to be the return value of the program, where usually zero indicates success and non-zero indicates an error.

Related topics

ABORT[1164] - ATEXIT[1165] - SYSTEM[1166]

1167

getenv

Syntax

```
include <cstdlib> char *getenv( const char *name );
```

The function getenv() returns environmental information associated with name, and is very implementation dependent. **NULL** is returned if no information about name is available.

Related topics

SYSTEM[1168]

1169

1164 Chapter 3.7.11 on page 356
1165 Chapter 3.7.11 on page 357
1166 Chapter 3.7.11 on page 365
1167 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
1168 Chapter 3.7.11 on page 365
1169 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

longjmp

Syntax

```
include <csetjmp> void longjmp( jmp_buf env, int val );
```

The function longjmp() behaves as a cross-function `goto` statement: it moves the point of execution to the record found in env, and causes setjmp() to return *val*. Using longjmp() may have some side effects with variables in the setjmp() calling function that were modified after the initial return.

longjmp() does not call destructors of any created objects. As such, it has been superseded with the C++ exception system, which uses the *throw* and *catch* keywords.

Related topics

SETJMP[1170]

1171

qsort

Syntax

```
include <cstdlib> void* qsort( const void *base, size_t num,
size_t size, int (*compare)(const void *, const void *));
```

The function qsort() performs a QUICK SORT[1172] on an array. Note that some implementations may instead use a more efficient sorting algorithm.

**base* refers to the array being sorted. This array contains *num* elements, each of size *size*.

1170 Chapter 3.7.11 on page 362
1171 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
1172 HTTP://EN.WIKIPEDIA.ORG/WIKI/QUICKSORT

The *compare* function accepts two pointers to the object within the array - which need to first be cast to the object type being examined. The function returns -1 if the first parameter should be before the second, 1 if the first parameter is after, or 0 if the object matches.

Related topics

BSEARCH[1173]

1174

raise

Syntax

```
include <csignal> int raise(int)
```

The raise() function raises a signal specified by its parameter.

If unsuccessful, it returns a non-zero value.

Related topics

SIGNAL[1175]

1176

rand

Syntax

```
include <cstdlib> int rand( void );
```

The function RAND[1177]() returns a pseudo-random integer between zero and **RAND_MAX**. An example:

1173 Chapter 3.7.11 on page 358
1174 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
1175 Chapter 3.7.11 on page 363
1176 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
1177 Chapter 3.7.11 on page 361

```
srand( time(NULL) );
for( i = 0; i < 10; i++ )
  printf( "Random number #%d: %d\n", i, rand() );
```

The rand() function must be seeded before its first call with the SRAND[1178]() function - otherwise it will consistently return the same numbers when the program is restarted.

Note:
The generation of **random numbers** is essential to CRYPTOGRAPHY[a]. Any STOCHASTIC PROCESS[b] (generation of random numbers) simulated by a computer, however, is not truly random, but pseudorandom; that is, the randomness of a computer is not from random radioactive decay of an unstable chemical isotope, but from predefined stochastic process, this is why this function needs to be seeded.

a HTTP://EN.WIKIBOOKS.ORG/WIKI/CRYPTOGRAPHY
b HTTP://EN.WIKIPEDIA.ORG/WIKI/STOCHASTIC%20PROCESS

Related topics

SRAND[1179]

1180

setjmp

Syntax

```
include <csetjmp> int setjmp( jmp_buf env );
```

The function setjmp() stores the current execution status in *env*, and returns 0. The execution state includes basic information about which code is being executed in preparation for the longjmp() function call. If and when longjmp is called, setjmp() will return the parameter provided by longjmp - however, on the second return, variables that were modified after the initial setjmp() call may have an undefined value.

1178 Chapter 3.7.11 on page 364
1179 Chapter 3.7.11 on page 364
1180 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

The buffer is only valid until the calling function returns, even if it is declared statically.

Since setjmp() does not understand constructors or destructors, it has been superseded with the C++ exception system, which uses the *throw* and *catch* keywords.

Note:
setjmp does not appear to be within the *std namespace*.

Related topics

LONGJMP[1181]

1182

signal

Syntax

```
include <csignal> void (*signal( int sig, void (*handler)(int)) )(int)
```

The signal() function takes two parameters - the first is the signal identifier, and the second is a function pointer to a signal handler that takes one parameter. The return value of signal is a function pointer to the previous handler (or SIG_ERR if there was an error changing the signal handler).

By default, most raised signals are handled either by the handlers SIG_DFL (which is the default signal handler that usually shuts down the program), or SIG_IGN (which ignores the signal and continues program execution.)

When you specify a custom handler and the signal is raised, the signal handler reverts to the default.

While the signal handlers are superseded by *throw* and *catch*, some systems may still require you to use these functions to handle some important events. For example, the signal SIGTERM on Unix-based systems indicates that the program should terminate soon.

1181 Chapter 3.7.11 on page 359
1182 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

Note:
List of standard signals in Solaris
SIGHUP, SIGINT, SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGEMT, SIGFPE, SIGKILL, SIGBUS, SIGSEGV, SIGSYS, SIGPIPE, SIGALRM, SIGTERM, SIGUSR1, SIGUSR2, SIGCHLD, SIGPWR, SIGWINCH, SIGURG, SIGIO, SIGSTOP, SIGTSTP, SIGCONT, SIGTTIN, SIGTTOU, SIGVTALRM, SIGPROF, SIGXCPU, SIGXFSZ, SIGWAITING, SIGLWP, SIGFREEZE, SIGTHAW, SIGCANCEL, SIGLOST

Related topics

RAISE[1183]

1184

srand

Syntax

```
include <cstdlib> void srand( unsigned seed );
```

The function srand() is used to seed the random sequence generated by RAND[1185](). For any given *seed*, RAND[1186]() will generate a specific "random" sequence over and over again.

```
srand( time(NULL) );
for( i = 0; i < 10; i++ )
  printf( "Random number #%d: %d\n", i, rand() );
```

Related topics

RAND[1187]
(Standard C Time & Date functions) TIME[1188]

1183 Chapter 3.7.11 on page 361
1184 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
1185 Chapter 3.7.11 on page 361
1186 Chapter 3.7.11 on page 361
1187 Chapter 3.7.11 on page 361
1188 Chapter 3.7.11 on page 352

1189

system

Syntax

```
include <cstdlib> int system( const char *command );
```

The system() function runs the given command by passing it to the default command interpreter.

The return value is usually zero if the command executed without errors. If command is **NULL**, system() will test to see if there is a command interpreter available. Non-zero will be returned if there is a command interpreter available, zero if not.

Related topics

EXIT[1190] - GETENV[1191]

1192

va_arg

Syntax

```
include <cstdarg> type va_arg( va_list argptr, type ); void va_-
end( va_list argptr ); void va_start( va_list argptr, last_parm );
```

The va_arg() macros are used to pass a variable number of arguments to a function.

1. First, you must have a call to va_start() passing a valid **va_list** and the mandatory first argument of the function. This first argument can be any-

1189 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
1190 Chapter 3.7.11 on page 358
1191 Chapter 3.7.11 on page 359
1192 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

thing; one way to use it is to have it be an integer describing the number of parameters being passed.
2. Next, you call va_arg() passing the **va_list** and the type of the argument to be returned. The return value of va_arg() is the current parameter.
3. Repeat calls to va_arg() for however many arguments you have.
4. Finally, a call to va_end() passing the **va_list** is necessary for proper cleanup.

```
int sum( int num, ... ) {
  int answer = 0;
  va_list argptr;

  va_start( argptr, num );

  for( ; num > 0; num-- ) {
    answer += va_arg( argptr, int );
  }

  va_end( argptr );

  return( answer );
}


int main( void ) {

  int answer = sum( 4, 4, 3, 2, 1 );
  printf( "The answer is %d\n", answer );

  return( 0 );
}
```

This code displays 10, which is 4+3+2+1.

Here is another example of variable argument function, which is a simple printing function:

```
void my_printf( char *format, ... ) {
  va_list argptr;

  va_start( argptr, format );

  while( *format != '\0' ) {
    // string
    if( *format == 's' ) {
      char* s = va_arg( argptr, char * );
      printf( "Printing a string: %s\n", s );
    }
    // character
    else if( *format == 'c' ) {
      char c = (char) va_arg( argptr, int );
      printf( "Printing a character: %c\n", c );
      break;
    }
```

```
    // integer
    else if( *format == 'd' ) {
      int d = va_arg( argptr, int );
      printf( "Printing an integer: %d\n", d );
    }

    *format++;
  }

  va_end( argptr );
}


int main( void ) {

  my_printf( "sdc", "This is a string", 29, 'X' );

  return( 0 );
}
```

This code displays the following output when run:

```
Printing a string: This is a string
Printing an integer: 29
Printing a character: X
```

1193

3.8 Debugging

Programming is a complex process, and since it is done by human beings, it often leads to errors. This makes debugging a fundamental skill of any programmer as debugging is an intrinsic part of programming.

For historical reasons, programming errors are called bugs (after an actual bug was found in a computer's mechanical relay, causing it to malfunction, as documented by Dr. Grace Hopper) and going through the code, examining it and looking for something wrong in the implementation (bugs) and correcting them is called debugging. The only help available to the programmer are the clues generated by the observable output. Other alternatives are running automated tools to test or verify the code or analyze the code as it runs, this is the task where a DEBUGGER[1194] can come to your aid.

1193 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
1194 HTTP://EN.WIKIPEDIA.ORG/WIKI/DEBUGGER

Debugging can be quite stressful, especially MULTI-THREADED[1195] programs that are extremely hard to debug, but it can also be a quite fun intellectual activity, kind of like a logic puzzle. Experience in debugging will not only reduce future errors but generate better hypothesis for what might be going wrong and ways to improve the design.

In debugging code there are already understood sections and situations that are prone to errors, for instance issues regarding pointer arithmetics is a well understood fragility inherited from C and in debugging, as any other methodology, there are already established techniques, procedures and practices that can make the detection of bugs easier (i.e.:DELTA DEBUGGING[1196]).

The field of debugging also covers establishing the security for the code (or the system it will run under). Of course this will all depend on the design limitations and requirements for the specific implementation.

3.8.1 Definition of bug

A bug in a program is defined by an unexpected behavior, unintended by the programmer. It happens when the behavior was not expected or intended in that program's code. A bug can also be described as error, flaw, mistake, FAILURE[1197], or FAULT[1198].

Most bugs arise from programming mistakes, and a few are caused by externalities (compiler, hardware or other systems outside of the direct responsibility of the programmer). A program that contains a large number of bugs, and/or bugs that seriously interfere with its functionality, is said to be **buggy**.

Reports detailing bugs in a program are commonly known as **bug reports**, fault reports, problem reports, trouble reports, change requests, and so forth.

There are a few different kinds of bugs that can occur in a program, and it is useful to distinguish between them in order to track them down more quickly.

Categorizations for bugs regarding their origin:

- **Organizational**

1195 Chapter 6.6.5 on page 609
1196 HTTP://EN.WIKIPEDIA.ORG/WIKI/DELTA%20DEBUGGING
1197 HTTP://EN.WIKIPEDIA.ORG/WIKI/FAILURE
1198 HTTP://EN.WIKIPEDIA.ORG/WIKI/FAULT%20%28TECHNOLOGY%29

  - Conceptual error. Where the code is syntactically correct, but the programmer or designer intended it to do something else. These can occur due to differences between the documentation and the actual product.
  - Unpropagated updates; e.g. programmer changes "myAdd" but forgets to change "mySubtract", which uses the same algorithm. These errors are mitigated by the DO NOT REPEAT YOURSELF[1199] philosophy.
  - Comments out of date or incorrect: many programmers assume the comments accurately describe the code.
- **External**
  - COMPILER BUGS[1200] or unexpected results due to lack of a default behavior on the C++ language specifications.
  - Environmental bugs on external dependencies (libraries or other software) or Operating System bugs/undocumented behaviors.
  - Hardware bugs or undocumented behaviors.
- **Arithmetic bugs**
  - DIVISION BY ZERO[1201].
  - ARITHMETIC OVERFLOW[1202] or UNDERFLOW[1203].
  - Loss of ARITHMETIC PRECISION[1204] due to ROUNDING[1205] or NUMERICALLY UNSTABLE[1206] algorithms.
- **Logic bugs**
  - INFINITE LOOP[1207]s and infinite RECURSION[1208].
  - OFF BY ONE ERROR[1209], counting one too many or too few when looping.
- **Syntax bugs** (TYPOS[1210])
- **Resource bugs**
  - NULL POINTER[1211] dereference.

1199 HTTP://EN.WIKIPEDIA.ORG/WIKI/DON%27T%20REPEAT%20YOURSELF
1200 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23COMPILER%20BUGS
1201 HTTP://EN.WIKIPEDIA.ORG/WIKI/DIVIDE%20BY%20ZERO%23DIVISION%20BY%20ZERO%20IN%20COMPUTER%20ARITHMETIC
1202 HTTP://EN.WIKIPEDIA.ORG/WIKI/ARITHMETIC%20OVERFLOW
1203 HTTP://EN.WIKIPEDIA.ORG/WIKI/ARITHMETIC%20UNDERFLOW
1204 HTTP://EN.WIKIPEDIA.ORG/WIKI/ARITHMETIC%20PRECISION
1205 HTTP://EN.WIKIPEDIA.ORG/WIKI/ROUNDING
1206 HTTP://EN.WIKIPEDIA.ORG/WIKI/NUMERICAL%20STABILITY
1207 HTTP://EN.WIKIPEDIA.ORG/WIKI/INFINITE%20LOOP
1208 HTTP://EN.WIKIPEDIA.ORG/WIKI/RECURSION%20%28COMPUTER%20SCIENCE%29
1209 HTTP://EN.WIKIPEDIA.ORG/WIKI/OFF%20BY%20ONE%20ERROR
1210 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23TYPOS
1211 HTTP://EN.WIKIPEDIA.ORG/WIKI/POINTER%20%28COMPUTING%29%23THE%20NULL%20POINTER

  - Using an UNINITIALIZED VARIABLE[1212].
  - Using an otherwise valid instruction on the wrong DATA TYPE[1213] (see PACKED DECIMAL[1214]/BINARY CODED DECIMAL[1215]).
  - ACCESS VIOLATION[1216]s.
  - Resource leaks, where a finite system resource such as MEMORY[1217] or FILE HANDLES[1218] are exhausted by repeated allocation without release.
  - BUFFER OVERFLOW[1219], in which a program tries to store data past the end of allocated storage. This may or may not lead to an access violation or STORAGE VIOLATION[1220]. These bugs can form a SECURITY VULNERABILITY[1221].
  - Excessive recursion which though logically valid causes STACK OVERFLOW[1222]
- **Co-processing bugs**
  - DEADLOCK[1223].
  - RACE CONDITION[1224].
  - Concurrency errors in CRITICAL SECTION[1225]s, MUTUAL EXCLUSION[1226]s and other features of CONCURRENT PROCESSING[1227]. TIME-OF-CHECK-TO-TIME-OF-USE[1228] (TOCTOU) is a form of unprotected critical section.

Common errors

Common programming errors are bugs mostly occur due to lack of experience, attention or when the programmer delegates too much responsibility to the compiler, IDE or other development tools.

1212 HTTP://EN.WIKIPEDIA.ORG/WIKI/UNINITIALIZED%20VARIABLE
1213 HTTP://EN.WIKIPEDIA.ORG/WIKI/DATA%20TYPE
1214 HTTP://EN.WIKIPEDIA.ORG/WIKI/PACKED%20DECIMAL
1215 HTTP://EN.WIKIPEDIA.ORG/WIKI/BINARY%20CODED%20DECIMAL
1216 HTTP://EN.WIKIPEDIA.ORG/WIKI/ACCESS%20VIOLATION
1217 HTTP://EN.WIKIPEDIA.ORG/WIKI/MEMORY%20LEAK
1218 HTTP://EN.WIKIPEDIA.ORG/WIKI/HANDLE%20LEAK
1219 HTTP://EN.WIKIPEDIA.ORG/WIKI/BUFFER%20OVERFLOW
1220 HTTP://EN.WIKIPEDIA.ORG/WIKI/STORAGE%20VIOLATION
1221 HTTP://EN.WIKIPEDIA.ORG/WIKI/SOFTWARE%20BUG%23SECURITY_VULNERABILITIES
1222 HTTP://EN.WIKIPEDIA.ORG/WIKI/STACK%20OVERFLOW
1223 HTTP://EN.WIKIPEDIA.ORG/WIKI/DEADLOCK
1224 HTTP://EN.WIKIPEDIA.ORG/WIKI/RACE%20CONDITION
1225 HTTP://EN.WIKIPEDIA.ORG/WIKI/CRITICAL%20SECTION
1226 HTTP://EN.WIKIPEDIA.ORG/WIKI/MUTUAL%20EXCLUSION
1227 HTTP://EN.WIKIPEDIA.ORG/WIKI/CONCURRENT%20PROGRAMMING%23COORDINATING%20ACCESS%20TO%20RESOURCES
1228 HTTP://EN.WIKIPEDIA.ORG/WIKI/TIME-OF-CHECK-TO-TIME-OF-USE

- Usage of uninitialized variables or pointers.
- Forgetting the differences between the debug and release version of the compiled code.
- Forgetting the `break` statement in a `switch` when fall-through was not meant
- Forgetting to check for null before accessing a member on a pointer.

```
// unsafe
p->doStuff();

// much better!
if (p)
{
   p->doStuff();
}
```

This will cause access violations (segmentation faults) and cause your program to halt unexpectedly.

Typos

Typos are a aggregation of simple to commit syntax errors (in very specific situations where the C++ language is ambivalent). The term comes from TYPOGRAPHICAL ERROR[1229] as in an error on the typing process.

Forgetting the ; at the end of a line. All time classic !

Use of the wrong operator, such as performing assignment instead of EQUALITY TEST[1230]. In simple cases often warned by the compiler.

```
// Example of an assignment of a number in an if statement when a comparison was
 meant.
if ( x = 143 ) // should be: if ( x == 143)
```

Forgetting the brackets in a multi lined loop or if statement.

```
if (x==3)
cout << x;
flag++;
```

1229 HTTP://EN.WIKIPEDIA.ORG/WIKI/TYPOGRAPHICAL%20ERROR
1230 HTTP://EN.WIKIPEDIA.ORG/WIKI/%3D%3D%23EQUALITY

Understanding the timing

Compile-time errors

The compiler can only translate a program if the program is syntactically correct; otherwise, the compilation fails and you will not be able to run your program. Syntax refers to the structure of your program and the rules about that structure.

For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a syntax error. So does this one

For most human readers, a few syntax errors are not a significant problem, which is why we can read the poetry of E. E. CUMMINGS[1231] without spewing error messages.

Compilers are not so forgiving. If there is a single syntax error anywhere in your program, the compiler will print an error message and quit, and you will not be able to run your program.

To make matters worse, there are more syntax rules in C++ than there are in English, and the error messages you get from the compiler are often not very helpful. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.

Linker errors

Most linker errors are generated when using improper settings on your compiler/IDE, most recent compilers will report some sort of information about the errors and if you keep in mind the linker function you will be able to easily address them. Most other sort of errors are due to improper use of the language or setup of the project files, that can lead to code collisions due to redefinitions or missing information.

Run-time errors

The run-time error, so-called because the error does not appear until you run the program.

Logic errors and semantics

1231 HTTP://EN.WIKIPEDIA.ORG/WIKI/E._E._CUMMINGS

The third type of error is the logical or semantic error. If there is a logical error in your program, it will compile and run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying logical errors can be tricky, since it requires you to work backwards by looking at the output of the program and trying to figure out what it is doing.

Compiler Bugs

As we have seen earlier, bugs are common to every programming task. Creating a compiler is no different, in fact creating a C++ compiler is an extremely complex programming task, more so since the language even if stable is always evolving and not only on the standard.

The liberty C++ permits enables programmers to push the envelop on what it is possible and expected and to an increase on the level of code complexity due to abstractions. This has lead to compilers to attempt to automating several low level actions to ease the burden to the programmer, like code optimization, higher level of interaction and control over the compiler components and the inclusion of very low level configuration possibilities. All these features increase the number of ways a compiler can end up generating incorrect (or sometimes technically correct but unexpected) results. The programmer should always keep in mind that compiler bugs are possible but extremely rare.

One of the most common bugs attributed to the compiler result from a badly configured optimization option (or an inability to understand them). If you suspect a compiler error turn optimizations off fist.

3.8.2 Experimental debugging

One of the most important skills you should acquire from working with this book is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways debugging is like detective work. You are confronted with clues and you have to infer the processes and events that lead to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was

correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As SHERLOCK HOLMES[1232] pointed out, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (from A. CONAN DOYLE'S[1233] The Sign of Four).

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should always start with a working program that does something, and make small modifications, debugging them as you go, so that you always have a working program.

For example, LINUX[1234] is an operating system that contains thousands of lines of code, but it started out as a simple program LINUS TORVALDS[1235] used to explore the Intel 80386 chip. According to Larry Greenfield, "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux" (from [ftp://sunsite.unc.edu//pub/Linux/docs/LDP/users-guide/!INDEX.html The Linux Users' Guide Beta Version 1], Page 10).

Endurance/Stress test

This sort of test is done to detect not only bugs but to mark opportunities for optimization. An **endurance test** is performed by analyzing multiple times the same actions as to gather statistical significant data. Note that this type of test is restricted to the selected set of actions and the projected variations, during the test, in regards to input processing.

Some automation is possible in this type of test, even dealing with simulating interaction with the users interface.

A **stress test** is a subtle variation of the endurance, the purpose is to determine and even establish the limits of the program as it processes inputs. Again the gathered metrics will only have significance in regards to the actions performed.

This tests and any variations will therefore depend on how they are designed and are extremely goal oriented, in the sense that they will only provide correct answerer to correctly asked questions. Reliance on results will have to be conser-

1232 HTTP://EN.WIKIPEDIA.ORG/WIKI/SHERLOCK_HOLMES
1233 HTTP://EN.WIKIPEDIA.ORG/WIKI/A._CONAN_DOYLE
1234 HTTP://EN.WIKIPEDIA.ORG/WIKI/LINUX
1235 HTTP://EN.WIKIPEDIA.ORG/WIKI/LINUS_BENEDICT_TORVALDS

vative, as the tester must acknowledge that some events may be absent from the scrutiny. This characteristic makes them more useful for optimization, since bottleneck in resource usage will provide a better starting point for analysis than for instance a crash or a deadlock.

3.8.3 Tracing

The technique of **TRACING**[1236] evolved directly from the hardware to the SOFTWARE ENGINEERING[1237] field. In field of hardware it consists on sampling the signals of an given circuit to verify the consistency of the hardware implemented logic/algorithm, as such earlier programmers adopted the term and function to trace the execution of the software with one particularly distinction, tracing should not be performed or enabled in public release versions.

There are several ways to execute the **tracing**, by simply include into the code report faculties that would produce the output of its state at run time (similarly to the errors and warnings the compiler and linker generates), one can even use the compiler and linker to report special messages. Another way is to interact directly to a debugger in a specified debug mode the debugger to interact with the running code. One can even integrate full fledged LOGGING[1238] systems that can record that same information in volume, and in an organized fashion, it all depends on the levels of complexity and detail required for the pertinent functionality one requires.

Event logging versus tracing

Logging can be an objective of a final product, but rarely covering the direct internal functioning of the main program, providing debug information useful for diagnostics and AUDITING[1239]. The debug information is typically only of interest to the programmers for debugging purposes, and additionally, depending on the type and detail of information contained in a trace log, by experienced SYSTEM ADMINISTRATOR[1240]s or TECHNICAL SUPPORT[1241] personnel to diagnose common problems with software. Tracing is a CROSS-CUTTING CONCERN[1242].

1236 HTTP://EN.WIKIPEDIA.ORG/WIKI/TRACING
1237 HTTP://EN.WIKIPEDIA.ORG/WIKI/SOFTWARE%20ENGINEERING
1238 HTTP://EN.WIKIPEDIA.ORG/WIKI/DATA%20LOGGING
1239 HTTP://EN.WIKIPEDIA.ORG/WIKI/AUDITING
1240 HTTP://EN.WIKIPEDIA.ORG/WIKI/SYSTEM%20ADMINISTRATOR
1241 HTTP://EN.WIKIPEDIA.ORG/WIKI/TECHNICAL%20SUPPORT
1242 HTTP://EN.WIKIPEDIA.ORG/WIKI/CROSS-CUTTING%20CONCERN

3.8.4 Debugger

Normally, there is no way to see the source code of a program while the program is running. This inability to "see under the covers" while the program is executing is a real handicap when you are debugging a program. The most primitive way of looking under the covers is to insert (depending on your programming language) print or display, or exhibit, or echo statements into your code, to display information about what is happening. But finding the location of a problem this way can be a slow, painful process. This is where a *debugger* comes in.

If you want to use a debugger and have never used one before, then you have two tasks ahead of you. Your first task is to learn basic debugger concepts and vocabulary. The second is to learn how to use the particular debugger that is available to you. The documentation for your debugger will help you with the second task, but it may not help with the first. In this section we will help you with the first task by providing an introduction to basic debugger concepts and terminology in regard to the language at hand. Once you become familiar with these basics, then your debugger's documentation/use should make more sense to you. Most software debugging is a slow manual process that does not scale well.

A *debugger* is a piece of software that enables you to run your program in debugging mode rather than in normal mode. Running a program in debugging mode allows you to look under the covers while your program is running. Specifically, a *debugger* enables you:

1. to see the source code of each statement in your program as that statement executes.
2. to suspend or pause execution of the program at places of your choosing.
3. while the program is paused, to issue various commands in order to examine and change the internal state of the program.
4. to resume (or continue) execution.

It is worth noting that there is a generally accepted set of debugger terms and concepts. Most debuggers are evolutionary descendants of a Unix console debugger for C named dbx, so they share concepts and terminology derived from dbx. Many visual debuggers are simply graphic wrappers around a console debugger, so visual debuggers share the same heritage, and the same set of concepts and terms. Programmers keep running into the same types of bugs that others have encountered (even across different languages by reusing code); one common example is buffer overruns.

Debuggers come in two flavors: **console-mode** (or simply console) debuggers and **visual** or **graphical** debuggers.

Console debuggers are often a part of the language itself, or included in the language's standard libraries. The user interface to a console debugger is the keyboard and a console-mode window (Microsoft Windows users know this as a "DOS console"). When a program is executing under a console debugger, the lines of source code stream past the console window as they are executed. A typical debugger has many ways to specify the exact places in the program where you want execution to pause. When the debugger pauses, it displays a special debugger prompt that indicates that the debugger is waiting for keyboard input. The user types in commands that tell the debugger what to do next. Typical commands would be to display the value of certain program variables, or to continue execution of the program.

Visual debuggers are typically available as one component of a multi-featured IDE (integrated development environment). A powerful and easy-to-use visual debugger is an important selling-point for an IDE. The user interface of a visual debugger typically looks like the interface of a graphical text editor. The source code is displayed on the screen, in much the same way that it is displayed when you are editing it. The debugger has its own toolbar or menu with specialized debugger features. And it may have a special debugger margin an area to the left of the source code, used for displaying symbols for breakpoints, the current-line pointer, and so on. As the debugger runs, some kind of visual pointer (perhaps a yellow arrow) will move down this debugger margin, indicating which statement has just finished executing, or which statement is about to be executed. Features of the debugger can be invoked by mouse-clicks on areas of the source code, the debugger margin, or the debugger menus.

How do you start the debugger?

How you start the debugger (or put your program into debugging mode) depends on your programming language and on the kind of debugger that you are using. If you are using a console debugger, then depending on the facilities offered by your particular debugger you may have a choice of several different ways to start the debugger. One way may be to add an argument (e.g. -d) to the command line that starts the program running. If you do this, then the program will be in debugging mode from the moment it starts running. A second way may be to start the debugger, passing it the name of your program as an argument. For example, if your debugger's name is `pdb` and your program's name is `myProgram`, then you might start executing your program by entering `pdb myProgram` at the command prompt. A third way may be to insert statements into the source code of your program statements that put your program into debugging mode. If you do this, when you start your program running, it will execute normally until it reaches the

debugging statements. When those statements execute, they put your program into debugging mode, and from that point on you will be in debugging mode.

If you are working with an IDE that provides a visual debugger, then there is usually a "debug" button or menu item on your toolbar. Clicking it will start your program running in debug mode. As the debugger runs, some kind of visual pointer will move down the debugger margin, indicating what statement is executing.

Tracing your program

To explore the features offered by debuggers, let us begin by imagining that you have a simple debugger to work with. This debugger is very primitive, with an extremely limited feature set. But as a purely hypothetical debugger, it has one major advantage over all real debuggers: simply wishing for a new feature causes that feature magically to be added to the debugger's feature set!

At the outset, your debugger has very few capabilities. Once you start the debugger, it will show you the code for one statement in your program, execute the statement, and then pause. When the debugger is paused, you can tell it to do only two things:

1. the command print <aVariableName> will print the value of a variable, and
2. the command step will execute the next statement and then pause again.

If the debugger is a console debugger, you must type these commands at the debugger prompt. If the debugger is a visual debugger, you can just click a Next button, or type a variable name into a special Show Variable window. And that is all the capabilities that the debugger has.

Although such a simple debugger is moderately useful, it is also very clumsy. Using it, you very quickly find yourself wishing for more control over where the debugger pauses, and for a larger set of commands that you can execute when the debugger is paused.

Controlling where the debugger pauses

What you desire most is for the debugger not to pause after every statement. Most programs do a lot of setup work before they get to the area where the real problems lie, and you are tired of having to step through each of those setup statements one statement at a time to get to the real trouble zone. In short, you wish you could *set breakpoints*. A *breakpoint* is an object that you can attach to a line of code. The

debugger runs without pausing until it encounters a line with a breakpoint attached to it. The breakpoint tells the debugger to pause, so the debugger pauses.

With breakpoint functionality added to the debugger (wishing for it has made it appear!), you can now set a breakpoint at the beginning of the section of the code where the problem lies, then start up the debugger. It will run the program until it reaches the breakpoint. Then it will pause, and you can start examining the situation with your print command.

But when you're finished using the print command, you are back to where you were before single-stepping through the remainder of the program with the step command. You begin to wish for an alternative to the step command for a run to next breakpoint command. With such a command, you can set multiple breakpoints in the program. Then, when you are paused at a breakpoint, you have the option of single-stepping through the code with the step command, or running to the next breakpoint with the run to next breakpoint command.

With our hypothetical debugger, wishing makes it so! Now you have on-the-fly control over where the program will pause next. You're starting to get some real control over the debugging process!

The introduction of the run to next breakpoint command starts you thinking. What other useful alternatives to the step command can you think of?

Often you find yourself paused at a place in the code where you know that the next 15 statements contain no problems. Rather than stepping through them one-by-one, you wish you could to tell the debugger something like step 15 and it would execute the next 15 statements before pausing.

When you are working your way through a program, you often come to a statement that makes a call to a subroutine. In such cases, the step command is in effect a step into command. That is, it drops down into the subroutine, and allows you to trace the execution of the statements inside the subroutine, one by one.

However, in many cases you know that there is no problem in the subroutine. In such cases, you want to tell the debugger to step over the subroutine call that is, to run the subroutine without pausing at any of the statements inside the subroutine. The step over command is a sort of step (but do not show me any of the messy details) command. (In some debuggers, the step over command is called next.)

When you use step or step into to drop down into a subroutine, it sometimes happens that you get to a point where there is nothing more in the subroutine that is of interest. You wish to be able to tell the debugger to step out or run until subroutine end, which would cause it to run without pause until it encountered a return statement (or an implicit return of control to its caller) and then to pause.

And you realize that the step over and step into commands might be useful with loops as well as with subroutines. When you encounter a looping construct (a `for` statement or a do while statement, for instance) it would be handy to be able to choose to step into or to step over the execution of the loop.

Almost always there comes a time when there is nothing more to be learned by stepping through the code. You wish for a command to tell the debugger to continue or simply run to the end of the program.

Even with all of these commands, if you are using a console debugger you find that you are still using the step command quite a bit, and you are getting tired of typing the word step. You wish that if you wanted to repeat a command, you could just hit the ENTER key at the debugger prompt, and the debugger would repeat the last command that you entered at the debugger prompt. Lo, wishing makes it so!

This is such a productivity feature, that you start thinking about other features that a console debugger might provide to improve its ease-of-use. You notice that you often need to print multiple variables, and you often want to print the same set of variables over and over again. You wish that you had some way to create a macro or alias for a set of commands. You might like, for example, to define a macro with an alias of foo the macro would consist of a set of debugger print statements. Once foo is defined, then entering foo at the debugger prompt runs the statements in the macro, just as if you had entered them at the debugger prompt.

Persistence

Eventually the end of the workday arrives. Your debugging work is not yet finished. You log off of your computer and go home for some well-earned rest. The next morning, you arrive at work bright-eyed and bushy-tailed and ready to continue debugging. You boot your computer, fire up the debugger, and find that all of the aliases, breakpoints, and watchpoints that you defined the previous day are gone! And now you have a really big wish for the debugger. You want it to have some persistence. You want it to be able to remember this stuff, so you do not have to re-create it every time you start a new debugger session.

You can define aliases at the debugger prompt, which is great for aliases that you need to invent for special occasions. But often, there is a set of aliases that you need in every debugging session. That is, you'd like to be able to save alias definitions, and automatically re-create the aliases when you start any debugging session.

Most debuggers allow you to create a file that contains alias definitions. That file is given a special name. When the debugger starts, it looks for the file with that special name, and automatically loads those alias definitions.

Examining the call stack

When you are stepping through a program, one of the questions that you may have is "How did I get to this point in the code?" The answer to this question lies in the *call stack* (also known as the *execution stack*) of the current statement. The *call stack* is a list of the functions that were entered to get you to your current statement. For example, if the main program module is MAIN, and MAIN calls function A, and function A calls function B, and function B calls function C, and function C contains statement S, then the execution stack to statement S is:

```
MAIN
    A
     B
      C
       statement S
```

In many interpreted languages, if your program crashes, the interpreter will print the call stack for you as a *stack trace*.

Conditional Breakpoints

Some debuggers allow you to attach a set of *conditions* to breakpoints. You may be able to specify that the debugger should pause at the breakpoint only if a certain condition is met (for example *VariableX > 100*) or if the value of a certain variable has changed since the last time the breakpoint was encountered. You may be able, for example, to set the breakpoint to break when a certain counter reaches a value of (say) 100. This would allow a loop to run 100 times before breaking.

A breakpoint that has conditions attached to it is called a *conditional breakpoint*. A breakpoint that has no conditions attached to it is called an *unconditional* or *simple breakpoint*. In some debuggers, *all* breakpoints have conditions attached to them, and "**unconditional**" breakpoints are simply breakpoints with a condition of *true*.

Watchpoints

Some debuggers support a kind of breakpoint called a *watch* or a *watchpoint*. A **watchpoint** is a *conditional breakpoint* that is not associated with any particular line, but with a variable. A watchpoint is useful when you would like to pause whenever a certain variable's value changes. Searching through your code, looking for every line that changes the variable's value, and setting breakpoints on those lines, would be both laborious and error-prone. Watchpoints allow you to avoid all of that by associating a breakpoint with a variable rather than a point in the source code. Once a watchpoint has been defined, then it "watches" its variable. Whenever the value of the variable changes, the code pauses and you will probably get a message telling you why execution has paused. Then you can look at where you are in the code and what the value of the variable is.

Setting Breakpoints in a Visual Debugger

How you create (or "set" or "insert") a breakpoint will depend on your particular debugger, and especially on whether it is a visual debugger or a console-mode debugger. In this section we discuss how you typically set breakpoints in a visual debugger, and in the next section we will discuss how it is done in a console-mode debugger.

Visual debuggers typically let you scroll through the code until you find a point where you want to set a breakpoint. You place the cursor on the line of where you want to insert the breakpoint and then press a special hotkey or click a menu item or icon on the debugger toolbar. If an icon is available, it may be something that suggests the act of watching for instance it may look like a pair of glasses or binoculars. At that point, a special dialog may pop up allowing you to specify whether the breakpoint is conditional or unconditional, and (if it is conditional) allowing you to specify the conditions associated with the breakpoint.

Once the breakpoint has been placed, many visual debuggers place a red dot or a red octagon (similar to a American/European traffic "STOP" SIGN[1243]) in the margin to indicate there is a breakpoint at that point in the code.

1243 HTTP://EN.WIKIPEDIA.ORG/WIKI/STOP_SIGN

3.8.5 Other runtime analyzers

3.9 Chapter Summary

1. THE CODE[1244] - includes list of recognized *keywords*[1245].
   a) FILE ORGANIZATION[1246]
   b) STATEMENTS[1247]
   c) CODING STYLE CONVENTIONS[1248]
   d) DOCUMENTATION[1249]
   e) SCOPE AND NAMESPACES[1250]
2. COMPILER[1251]
   a) PREPROCESSOR[1252] - includes *the* STANDARD HEADERS[1253].
   b) LINKER[1254]
3. VARIABLES AND STORAGE[1255] - *locality*, *scope* and *visibility*, including SOURCE EXAMPLES[1256].
   a) TYPE[1257]
4. OPERATORS[1258] - *precedence order* and *composition*, , *assignment*, `sizeof`, **new**, **delete**, **[]** (*arrays*[1259]), ***** (*pointers*[1260]) and **&** (*references*).
   a) LOGICAL OPERATORS[1261] - the **&&** (and), || (or), and **!** (not).
   b) CONDITIONAL OPERATOR[1262] - the **?:**
5. TYPE CASTING[1263] - *Automatic*, *explicit* and *advanced type casts*.

1244 Chapter 3 on page 41
1245 Chapter 3.1.3 on page 46
1246 Chapter 3.1.5 on page 49
1247 Chapter 3.1.6 on page 56
1248 Chapter 3.1.7 on page 59
1249 Chapter 3.1.8 on page 74
1250 Chapter 3.1.9 on page 78
1251 Chapter 3.1.10 on page 87
1252 Chapter 3.2.2 on page 98
1253 Chapter 3.2.3 on page 100
1254 Chapter 3.2.3 on page 117
1255 Chapter 3.2.4 on page 121
1256 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FCODE%2FVARIABLES%2FEXAMPLES
1257 Chapter 3.3.3 on page 138
1258 Chapter 3.3.4 on page 163
1259 Chapter 3.4.10 on page 178
1260 Chapter 3.4.10 on page 184
1261 Chapter 3.4.12 on page 200
1262 Chapter 3.4.13 on page 203
1263 Chapter 3.4.14 on page 204

6. FLOW OF CONTROL[1264] - Conditionals (**if**, **if**-**else**, **switch**), loop iterations (**while**, **do**-**while**, `for`) and `goto`.
7. FUNCTIONS[1265] - Introduction (including **main**), *argument passing*, *returning values*, *recursive functions*, *pointers to functions* and *function overloading*.
   a) STANDARD C LIBRARY[1266] - I/O[1267], STRING AND CHARACTER[1268], MATH[1269], TIME AND DATE[1270], MEMORY[1271] and OTHER STANDARD C FUNCTIONS[1272]
8. DEBUGGING[1273] - Finding, fixing, preventing bugs and using debugging tools.

2[1274]

2[1275]

1264 Chapter 3.5.2 on page 213
1265 Chapter 3.6.3 on page 229
1266 Chapter 3.7.10 on page 264
1267 Chapter 3.7.11 on page 273
1268 Chapter 3.7.11 on page 303
1269 Chapter 3.7.11 on page 330
1270 Chapter 3.7.11 on page 345
1271 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FCODE%2FSTANDARD%20C%20LIBRARY%2FMEMORY%20
1272 Chapter 3.7.11 on page 356
1273 Chapter 3.7.11 on page 367
1274 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
1275 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

4 Object Oriented Programming

4.1 Structures

A simple implementation of the object paradigm from (OOP) that holds collections of data records (also known as *compound values* or *set*). A `struct` is like a class **except for the default access** (class has default access of private, struct has default access of public). C++ also guarantees that a struct that only contains C types is equivalent to the same C struct thus allowing access to legacy C functions, it can (but may not) also have constructors (and must have them, if a templated class is used inside a `struct`), as with Classes the compiler implicitly-declares a destructor if the struct doesn't have a user-declared destructor. Structures will also allow OPERATOR OVERLOADING[1].

A struct is *defined* by:

```
struct myStructType /*: inheritances */ {
public:
 // public members
protected:
 // protected members
private:
 // private members
} myStructName;
```

Because it is not supported in C, it is uncommon to have structs in C++ using inheritances even though they are supported just like in classes. The more distinctive aspect is that structs can have two identities one is in reference to the type and another to the specific object. The public access label can sometimes be ignored since the default state of struct for member functions and fields is public.

An object of type *myStructType* (case-sensitive) is *declared* using:

```
myStructType obj1;
```

1 Chapter 4.6 on page 438

Note:
From a technical viewpoint, a struct and a class are practically the same thing. A struct can be used anywhere a class can be and vice-versa, the only technical difference is that class members default to *private* and struct members default to *public*. Structs can be made to behave like classes simply by putting in the keyword private at the beginning of the struct. Other than that it is mostly a difference in convention.

Why should you Use Structs, Not Classes?

Older programmer languages used a similar type called Record (i.e.: COBOL, FORTRAN) this was implemented in C as the struct keyword. And so C++ uses structs to comply with this C's heritage (the code and the programmers). Structs are simpler to be managed by the programmer and the compiler. One should use a `struct` for POD (PLAINOLDDATA[2]) types that have no methods and whose data members are all `public`. `struct` may be used more efficiently in situations that default to public inheritance (which is the most common kind) and where `public` access (which is what you want if you list the public interface first) is the intended effect. Using a `class`, you typically have to insert the keyword public in two places, for no real advantage. In the end it's just a matter of convention, which programmers should be able to get used to.

Point objects

As a simple example of a compound structure, consider the concept of a mathematical point. At one level, a point is two numbers (coordinates) that we treat collectively as a single object. In mathematical notation, points are often written in parentheses, with a comma separating the coordinates. For example, (0, 0) indicates the origin, and (x, y) indicates the point x units to the right and y units up from the origin.

The natural way to represent a point is using two doubles. The **structure** or `struct` is one of the solutions to group these two values into a compound object.

```
// A struct definition:
 struct Point {
   double x, y; };
```

2 HTTP://EN.WIKIBOOKS.ORG/WIKI/WIKI%3APLAINOLDDATA

This definition indicates that this structure contains two members, named x and y. These members are also called instance variables, for reasons I will explain a little later.

It is a common error to leave off the semi-colon at the end of a structure definition. It might seem odd to put a semi-colon after a squiggly-brace, but you'll get used to it. This syntax is in place to allow the programmer the facility to create an instance[s] of the struct when it is defined.

Once you have defined the new structure, you can create variables with that type:

```
struct Point blank;
blank.x = 3.0;
blank.y = 4.0;
```

The first line is a conventional variable declaration: blank has type Point. The next two lines initialize the instance variables of the structure. The "dot notation" used here is similar to the syntax for invoking a function on an object, as in `fruit.length()`. Of course, one difference is that function names are always followed by an argument list, even if it is empty.

As usual, the name of the variable blank appears outside the box and its value appears inside the box. In this case, that value is a compound object with two named instance variables.

Accessing instance variables

You can read the values of an instance variable using the same syntax we used to write them:

```
int x = blank.x;
```

The expression `blank.x` means "go to the object named blank and get the value of the member named x." In this case we assign that value to a local variable named x. Notice that there is no conflict between the local variable named x and the instance variable named x. The purpose of dot notation is to identify which variable you are referring to unambiguously.

You can use dot notation as part of any expression, so the following are legal.

```
cout << blank.x << ", " << blank.y << endl;
double distance = sqrt(blank.x * blank.x + blank.y * blank.y);
```

The first line outputs 3, 4; the second line calculates the value 5.

Operations on structures

Most of the operators we have been using on other types, like mathematical operators (+, %, etc.) and comparison operators (==, >, etc.), do not work on structures. Actually, it is possible to define the meaning of these operators for the new type, but we won't do that in this book.

On the other hand, the assignment operator does work for structures. It can be used in two ways: to initialize the instance variables of a structure or to copy the instance variables from one structure to another. An initialization looks like this:

```
Point blank = { 3.0, 4.0 };
```

The values in curly brackets get assigned to the instance variables of the structure one by one, in order. So in this case, x gets the first value and y gets the second.

Unfortunately, this syntax can be used only in an initialization, not in an assignment statement. Therefore, the following is illegal.

```
Point blank;
blank = { 3.0, 4.0 }; // WRONG !!
```

You might wonder why this perfectly reasonable statement should be illegal, and there is no good answer. (Note, however, that a *similar* syntax is legal in C since 1999, and is under consideration for possible inclusion in C++ in the future.)

On the other hand, it is legal to assign one structure to another. For example:

```
Point p1 = { 3.0, 4.0 };
Point p2 = p1;
cout << p2.x << ", " <<  p2.y << endl;
```

The output of this program is `3, 4`.

Structures as return types

You can write functions that return structures. For example, findCenter takes a Rectangle as an argument and returns a Point that contains the coordinates of the center of the Rectangle:

```
Point findCenter (Rectangle& box)
{
  double x = box.corner.x + box.width/2;
  double y = box.corner.y + box.height/2;
  Point result = {x, y};
```

```
  return result;
}
```

To call this function, we have to pass a box as an argument (notice that it is being passed by reference), and assign the return value to a Point variable:

```
Rectangle box = { {0.0, 0.0}, 100, 200 };
Point center = findCenter (box);
printPoint (center);
```

The output of this program is (50, 100).

Passing other types by reference

It's not just structures that can be passed by reference. All the other types we've seen can, too. For example, to swap two integers, we could write something like:

```
void swap (int& x, int& y)
{
  int temp = x;
  x = y;
  y = temp;
}
```

We would call this function in the usual way:

```
int i = 7;
int j = 9;
swap (i, j);
cout << i << j << endl;
```

The output of this program is 97. Draw a stack diagram for this program to convince yourself this is true. If the parameters x and y were declared as regular parameters (without the &s), swap would not work. It would modify x and y and have no effect on i and j.

When people start passing things like integers by reference, they often try to use an expression as a reference argument. For example:

```
int i = 7;
int j = 9;
swap (i, j+1); // WRONG!!
```

This is not legal because the expression `j+1` is not a variable — it does not occupy a location that the reference can refer to. It is a little tricky to figure out exactly what kinds of expressions can be passed by reference. For now, a good rule of thumb is that reference arguments have to be variables.

Pointers and structures

Structures can also be pointed by pointers and store pointers. The rules are the same as for any fundamental data type. The pointer must be declared as a pointer to the structure.

4.1.1 Nesting structures

Structures can also be nested so that a valid element of a structure can also be another structure.

```
//of course you have to define the Point struct first!

struct Rectangle {
 Point upper_left;
 Point upper_right;
 Point lower_left;
 Point lower_right;
};
```

4.1.2 this

The **this** keyword is an implicitly created pointer that is only accessible within nonstatic member functions of a struct (or a union or class) and points to the object for which the member function is called. This pointer is not available in static member functions. This will be restated again on when introducing unions a more in depth analysis is provided in the SECTION ABOUT CLASSES[3].

4.2 union

The **union** keyword is used to define a union type.

Syntax

```
union union-name {
public-members-list;
```

3 Chapter 4.3.4 on page 405

```
    private:
    private-members-list;
    } object-list;
```

Union is similar to `struct` (more that `class`), unions differ in the aspect that the fields of a `union` share the same position in memory and are by default `public` rather than `private`. The size of the `union` is the size of its largest field (or larger if alignment so requires, for example on a SPARC machine a `union` contains a `double` and a `char [17]` so its size is likely to be 24 because it needs 64-bit alignment). Unions cannot have a `destructor`.

What is the point of this? Unions provide multiple ways of viewing the same memory location, allowing for more efficient use of memory. Most of the uses of unions are covered by object-oriented features of C++, so it is more common in C. However, sometimes it is convenient to avoid the formalities of object-oriented programming when performance is important or when one knows that the item in question will not be extended.

```
union Data {
  int i;
  char c;
};
```

4.2.1 Writing to Different Bytes

Unions are very useful for low-level programming tasks that involve writing to the same memory area but at different portions of the allocated memory space, for instance:

```
union item {
  // The item is 16-bits
  short theItem;
  // In little-endian lo accesses the low 8-bits -
  // hi, the upper 8-bits
  struct { char lo; char hi; } portions;
};
```

Note:
A name for the struct declared in item can be omitted because it is not used. All that needs to be explicitly named is the parts that we intend to access, namely the instance itself, portions.

```
  item tItem;
```

```
  tItem.theItem = 0xBEAD;
  tItem.portions.lo = 0xEF; // The item now equals 0xBEEF
```

Using this union we can modify the low-order or high-order bytes of theItem without disturbing any other bytes.

4.2.2 Example in Practice: SDL Events

One real-life example of unions is the event system of SDL, a graphics library in C. In graphical programming, an event is an action triggered by the user, such as a mouse move or keyboard press. One of the SDL's responsibilities is to handle events and provide a mechanism for the programmer to listen for and react to them.

Note:
The following section deals with a library in C rather than C++, so some features, such as methods of objects, are not used here. However C++ is more-or-less a superset of C, so you can understand the code with the knowledge you have gained so far.

```
// primary event structure in SDL

typedef union {
  Uint8 type;
  SDL_ActiveEvent active;
  SDL_KeyboardEvent key;
  SDL_MouseMotionEvent motion;
  SDL_MouseButtonEvent button;
  SDL_JoyAxisEvent jaxis;
  SDL_JoyBallEvent jball;
  SDL_JoyHatEvent jhat;
  SDL_JoyButtonEvent jbutton;
  SDL_ResizeEvent resize;
  SDL_ExposeEvent expose;
  SDL_QuitEvent quit;
  SDL_UserEvent user;
  SDL_SysWMEvent syswm;
} SDL_Event;
```

Each of the types other than `Uint8` (an 8-bit `unsigned` integer) is a struct with details for that particular event.

```
// SDL_MouseButtonEvent

typedef struct{
  Uint8 type;
  Uint8 button;
  Uint8 state;
```

```
  Uint16 x, y;
} SDL_MouseButtonEvent;
```

When the programmer receives an event from SDL, he first checks the type value. This tells him what kind of an event it is. Based on this value, he either ignores the event or gets more information by getting the appropriate part of the union.

For example, if the programmer received an event in `SDL_Event ev`, he could react to mouse clicks with the following code.

```
if (ev.type == SDL_MOUSEBUTTONUP && ev.button.button == SDL_BUTTON_RIGHT) {
  cout << "You have right-clicked at coordinates (" << ev.button.x << ", "
       << ev.button.y << ")." << endl;
}
```

Note:
As each of the SDL_SomethingEvent structs contain a `Uint8 type` entry, it is safe to access both `Uint8 type` and the corresponding sub-struct together.

While identical functionality can be provided with a struct rather than a union, the union is far more space efficient; the struct would use memory for each of the different event types, whereas the union only uses memory for one. As only one entry has meaning per instance, it is reasonable to use a union in this case.

This scheme could also be constructed with polymorphism and inheritance features of object-oriented C++, however the setup would be involved and less efficient than this one. Use of unions loses type safety, however it gains in performance.

4.2.3 this

The this keyword is a implicitly created pointer that is only accessible within non-static member functions of a union (or a struct or class) and points to the object for which the member function is called. The this pointer is not available in static member functions. This will be restated again on when introducing unions a more in depth analysis is provided in the SECTION ABOUT CLASSES[4].

5

4 Chapter 4.3.4 on page 405

5 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

4.3 Classes

Classes are used to create *user defined types*. An instance of a class is called an *object* and programs can contain any number of classes. As with other types, object types are case-sensitive.

Classes provide *encapsulation* as defined in the Object Oriented Programming (OOP) paradigm. A class can have both data members and functions members associated with it. Unlike the built-in types, the class can contain several variables and functions, those are called members.

Classes also provide flexibility in the "DIVIDE AND CONQUER[6]" scheme in program writing. In other words, one programmer can write a class and guarantee an interface. Another programmer can write the main program with that expected interface. The two pieces are put together and compiled for usage.

Note:
From a technical viewpoint, a struct and a class are practically the same thing. A struct can be used anywhere a class can be and vice-versa, the only technical difference is that class members default to *private* and struct members default to *public*. Structs can be made to behave like classes simply by putting in the keyword private at the beginning of the struct. Other than that it is mostly a difference in convention.
The C++ standard does not have a definition for *method*. When discussing with users of other languages, the use of the word *method* to represent a member function can at times become confusing or raise problems to interpretation, like referring to a static member function as a static method. It is even common for some C++ programmers to use the term method to refer specifically to a virtual member functions in an informal context.

4.3.1 Declaration

A class is *defined* by:

```
class MyClass
{
 /* public, protected and private
 variables, constants, and functions */
};
```

6 HTTP://EN.WIKIPEDIA.ORG/WIKI/DIVIDE%20AND%20CONQUER

An object of type *MyClass* (case-sensitive) is *declared* using:

```
MyClass object;
```

- by default, all class members are initially *private*.
- keywords *public* and *protected* allow access to class members.
- classes contain not only data members, but also functions to manipulate that data.
- a class is used as the basic building block of OOP (this is a distinction of convention, not of language-enforced semantics).

A class can be created

- before main() is called.
- when a function is called in which the object is declared.
- when the "new" operator is used.

Class Names

- Name the class after what it is. If you can't determine a name, then you have not designed the system well enough.
- Compound names of over three words are a clue your design may be confusing various entities in your system. Revisit your design. Try a CRC card session to see if your objects have more responsibilities than they should.
- Avoid the temptation of naming a class something similar to the class it is derived from. A class should stand on its own. Declaring an object with a class type doesn't depend on where that class is derived from.
- Suffixes or prefixes are sometimes helpful. For example, if your system uses agents then naming something DownloadAgent conveys real information.

Data Abstraction

A fundamental concept of Object Oriented (OO) recommends an object should not expose any of its implementation details. This way, you can change the implementation without changing the code that uses the object. The class, by design, allows its programmer to hide (and also prevents changes as to) how the class is implemented. This powerful tool allows the programmer to build in a 'preventive' measure. Variables within the class often have a very significant role in what the class does, therefore variables can be secured within the *private* section of the class.

4.3.2 Access labels

The access labels **Public**, **Protected** and **Private** are used within classes to set access permissions for the members in that section of the *class*. All class members are initially *private* by default. The labels can be in any order. These labels can be used multiple times in a class declaration for cases where it is logical to have multiple groups of these types. An access label will remain active until another access label is used to change the permissions.

We have already mentioned that a class can have member functions "inside" it; we will see more about them later. Those member functions can access and modify all the data and member function that are inside the class. Therefore, permission labels are to restrict access to member function that reside outside the class and for other classes.

For example, a class "Bottle" could have a private variable *fill*, indicating a liquid level 0-3 dl. *fill* cannot be modified directly (compiler error), but instead *Bottle* provides the member function sip() to reduce the liquid level by 1. Mywaterbottle could be an instance of that class, an object.

```
/* Bottle - Class and Object Example */
#include <iostream>
#include <iomanip>

using namespace std;

class Bottle
{
  private:      // variables are modified by member functions of class
  int iFill;    // dl of liquid

  public:
    Bottle()    // Default Constructor
    : iFill(3)  // They start with 3 dl of liquid
      {
        // More constructor code would go here if needed.
      }

     bool sip() // return true if liquid was available
     {

       if (iFill > 0)
       {
         --iFill;
         return true;
       }
       else
       {
         return false;
       }
```

```
    }

    int level() const  // return level of liquid dl
    {
        return iFill;
    }
};  // Class declaration has a trailing semicolon

int main()
{
  // terosbottle object is an instance of class Bottle
  Bottle terosbottle;
  cout << "In the beginning, mybottle has "
       << terosbottle.level()
       << "  dl of liquid"
       << endl;

  while (terosbottle.sip())
  {
     cout << "Mybottle has "
          << terosbottle.level()
          << " dl of liquid"
          << endl;
  }

  return 0;
}
```

These keywords, *private, public, and protected,* affect the permissions of the members -- whether functions or variables.

public

This label indicates any members within the 'public' section can accessed freely anywhere a declared object is in scope.

> **Note:**
> Avoid declaring public data members, since doing so would contribute to create unforeseen disasters.

private

Members defined as private are only accessible within the class defining them, or friend classes. Usually the domain of member variables and helper functions. It's often useful to begin putting functions here and then moving them to the higher access levels as needed so to reduce complexity.

Note:
It's often overlooked that different instances of the same class may access each others' private or protected variables. A common case for this is in copy constructors.

(This is an example where the default copy constructor will do the same thing.)

```
class Foo
{
 public:
   Foo(const Foo &f)
   {
     m_iValue = f.m_iValue; // perfectly legal
   }

 private:
   int m_iValue;
};
```

protected

The protected label has a special meaning to inheritance, protected members are accessible in the class that defines them and in classes that inherit from that base class, or friends of it. In the section on inheritance we will see more about it.

Note:
Other instances of the same class can access a protected field - provided the two classes are of the same type. However, an instance of a child class cannot access a protected field or method of an instance of a parent class.

4.3.3 Inheritance (Derivation)

As we have seen early as we introduced PROGRAMMING PARADIGMS[7], **INHERITANCE**[8] is a property that describes a relationship between two (or more) types, or classes, of objects in OOP and C++ classes share this property. This in it self in not an abstraction but a characteristic of OOP.

7 Chapter 2.2.3 on page 16
8 Chapter 2.3.4 on page 20

Derivation is the action of creating a new class using the inheritance property of the C++ programming language. It is possible to derive one class from another or even several (**MULTIPLE INHERITANCE**[9]), like a tree we can call base class to the root and child class to any leaf; in any other case the parent/child relation will exist for each class derived from another.

Base Class

A base class is a class that is created with the intention of deriving other classes from it.

Child Class

A child class is a class that was derived from another, that will now be the parent class to it.

Parent Class

A parent class is the closest class that we derived from to create the one we are referencing as the child class.

As an example, suppose you are creating a game, something using different cars, and you need specific type of car for the policemen and another type for the player(s). Both car types share similar properties. The major difference (on this example case) would be that the policemen type would have sirens on top of their cars and the players' cars will not.

One way of getting the cars for the policemen and the player ready is to create separate classes for policemen's car and for the player's car like this:

```
class PlayerCar {
private:
  int color;

public:
  void driveAtFullSpeed(int mph){
    // code for moving the car ahead
  }

};
```

9 Chapter 4.3.3 on page 403

```
class PoliceCar {
private:
  int color;
  bool sirenOn;  // identifies whether the siren is on or not
  bool inAction; // identifies whether the police is in action (following the
 player) or not

public:
  bool isInAction(){
    return this->inAction;
  }

  void driveAtFullSpeed(int mph){
    // code for moving the car ahead
  }

};
```

and then creating separate objects for the two cars like this:

```
PlayerCar player1;
PoliceCar policemen1;
```

So, except for one thing that you can easily notice: there are certain parts of code that are very similar (if not exactly the same) in the above two classes. In essence, you have to type in the same code at two different locations! And when you update your code to include methods (functions) for `handBrake()` and `pressHorn()`, you'll have to do that in both the classes above.

Therefore, to escape this frustrating (and confusing) task of writing the same code at multiple locations in a single project, you use Inheritance.

Now that you know what kind of problems Inheritance solves in C++, let's examine how to implement Inheritance in our programs. As its name suggests, Inheritance lets us create new classes which automatically have all the code from existing classes. It means that if there is a class called `MyClass`, a new class with the name `MyNewClass` can be created which will have all the code present inside the `MyClass` class. The following code segment shows it all:

```
class MyClass {
  protected:
        int age;
  public:
        void sayAge(){
            this->age = 20;
            cout << age;
        }
};

class MyNewClass : public MyClass {
```

```
};

int main() {

  MyNewClass *a = new MyNewClass();
  a->sayAge();

  return 0;

}
```

As you can see, using the colon ':' we can inherit a new class out of an existing one. It's that simple! All the code inside the `MyClass` class is now available to the `MyNewClass` class. And if you are intelligent enough, you can already see the advantages it provides. If you are like me (i.e. not too intelligent), you can see the following code segment to know what I mean:

```
class Car {
  protected:
        int color;
        int currentSpeed;
        int maxSpeed;
  public:
        void applyHandBrake(){
            this->currentSpeed = 0;
        }
        void pressHorn(){
            cout << "Teeeeeeeeeeeeent"; // funny noise for a horn
        }
        void driveAtFullSpeed(int mph){
             // code for moving the car ahead;
        }
};

class PlayerCar : public Car {

};

class PoliceCar : public Car {
  private:
        bool sirenOn;  // identifies whether the siren is on or not
        bool inAction; // identifies whether the police is in action (following
 the player) or not
  public:
        bool isInAction(){
            return this->inAction;
        }
};
```

In the code above, the two newly created classes *PlayerCar* and *PoliceCar* have been inherited from the *Car* class. Therefore, all the methods and properties (variables) from the *Car* class are available to the newly created classes for the player's car and the policemen's car. Technically speaking, in C++, the *Car* class in this

case is our "Base Class" since this is the class which the other two classes are based on (or inherit from).

Just one more thing to note here is the keyword *protected* instead of the usual *private* keyword. That's no big deal: We use *protected* when we want to make sure that the variables we define in our base class should be available in the classes that inherit from that base class. If you use *private* in the class definition of the *Car* class, you will not be able to inherit those variables inside your inherited classes.

There are three types of class inheritance: public, private and protected. We use the keyword *public* to implement public inheritance. The classes who inherit with the keyword public from a base class, inherit all the public members as public members, the protected data is inherited as protected data and the private data is inherited but it cannot be accessed directly by the class.

The following example shows the class Circle that inherits "publicly" from the base class Form:

```
class Form {
private:
  double area;

public:
  int color;

  double getArea(){
    return this->area;
  }

  void setArea(double area){
    this->area=area;
  }

};

class Circle: public Form {
public:
  double getRatio() {
    double a;
    a= getArea();
    return sqrt(a/2*3.14);
  }

  void setRatio(double diameter) {
    setArea( pow( diameter * 0.5, 2) * (3.14));
  }

  bool isDark() {
    return color>10;
  }

};
```

The new class Circle inherits the attribute area from the base class Form (the attribute area is implicitly an attribute of the class Circle), but it cannot access it directly. It does so through the functions getArea and setArea (that are public in the base class and remain public in the derived class). The color attribute, however, is inherited as a public attribute, and the class can access it directly.

The following table indicates how the attributes are inherited in the three different types of inheritance:

	private	protected	public
private inheritance	The member is inaccessible.	The member is private.	The member is private.
protected inheritance	The member is inaccessible.	The member is protected.	The member is protected.
public inheritance	The member is inaccessible.	The member is protected.	The member is public.

As the table above shows, protected members are inherited as protected methods in public inheritance. Therefore, we should use the protected label whenever we want to declare a method inaccessible outside the class and not to lose access to it in derived classes. However, losing accessibility can be useful sometimes, because we are encapsulating details in the base class.

Let's imagine that we have a class with a very complex method "m" that invokes many auxiliary methods declared as private in the class. If we derive a class from it, we should not bother about those methods because they are inaccessible in the derived class. If a different programmer is in charge of the design of the derived class, allowing access to those methods could be the cause of errors and confusion. So, it is a good idea to avoid the protected label whenever we can have a design with the same result with the private label.

Multiple inheritance

MULTIPLE INHERITANCE[10] allows the construction of classes that inherit from more than one type or class. This contrasts with single inheritance, where a class will only inherit from one type or class.

Multiple inheritance can cause some confusing situations, and is much more complex than single inheritance, so there is some debate over whether or not its benefits

10 Chapter 2.3.4 on page 21

outweigh its risks. Multiple inheritance has been a touchy issue for many years, with opponents pointing to its increased complexity and ambiguity in situations such as the "DIAMOND PROBLEM[11]". Most modern OOP languages do not allow multiple inheritance.

The declared order of derivation is relevant for determining the order of default initialization by constructors and destructors cleanup.

```
class One
{
  // class internals
}

class Two
{
  // class internals
}

class MultipleInheritance : public One, public Two
{
  // class internals
}
```

Note:
Remember that when creating classes that will be derived from, the destructor may require further considerations.

12

4.3.4 Data members

Data members are declared in the same way as a global or function variable, but as part of the class definition. Their purpose is to store information for that class and may include members of any type, even other user-defined types. They are usually hidden from outside use, depending on the coding style adopted, external use is normally done through SPECIAL MEMBER FUNCTIONS[13].

11 HTTP://EN.WIKIPEDIA.ORG/WIKI/DIAMOND%20PROBLEM
12 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
13 Chapter 4.3.1 on page 394

Note:
Explicit initializers are not allowed inside the class definition, except if they are `const static int` or enumeration types, these may have an explicit initializer.

this pointer

The *this* keyword acts as a pointer to the class being referenced. The *this* pointer acts like any other pointer, although you can't change the pointer itself. Read the section concerning POINTERS AND REFERENCES[14] to understand more about general pointers.

The *this* pointer is only accessible within nonstatic member functions of a **class**, **union** or **struct**, and is not available in static member functions. It is not necessary to write code for the *this* pointer as the compiler does this implicitly. When using a debugger, you can see the *this* pointer in some variable list when the program steps into nonstatic class functions.

In the following example, the compiler inserts an implicit parameter *this* in the nonstatic member function int getData(). Additionally, the code initiating the call passes an implicit parameter (provided by the compiler).

```
class Foo
{
private:
    int iX;
public:
    Foo(){ iX = 5; };

    int getData()
    {
        return this->iX;  // this is provided by the compiler at compile time
    }
};

int main()
{
    Foo Example;
    int iTemp;

    iTemp = Example.getData(&Example);  // compiler adds the &Example reference
 at compile time
```

14 Chapter 3.4.1 on page 164

```
    return 0;
}
```

There are certain times when a programmer should know about and use the *this* pointer. The *this* pointer should be used when overloading the assignment operator to prevent a catastrophe. For example, add in an assignment operator to the code above.

```
class Foo
{
private:
    int iX;
public:
    Foo() { iX = 5; };

    int getData()
    {
        return iX;
    }

    Foo& operator=(const Foo &RHS);
};

Foo& Foo::operator=(const Foo &RHS)
{
    if(this != &RHS)
    {    // the if this test prevents an object from copying to itself (ie. RHS =
 RHS;)
        this->iX = RHS.iX;     // this is suitable for this class, but can be
 more complex when
                               // copying an object in a different much larger
 class
    }

    return (*this);           // returning an object allows chaining, like a = b
 = c; statements
}
```

However little you may know about *this*, it is important in implementing any class.

15

static data member

The use of the `static` specifier in a data member, will cause that member to be shared by all instances of the owner class and derived classes. To use static data members you must declare the data member as static and initialize it outside of the class declaration, at file scope.

15 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

When used in a class data member, all instantiations of that class share one copy of the variable.

```
class Foo {
public:
  Foo() {
    ++iNumFoos;
    cout << "We have now created " << iNumFoos << " instances of the Foo
 class\n";
  }
private:
  static int iNumFoos;
};

int Foo::iNumFoos = 0;  // allocate memory for numFoos, and initialize it

int main() {
  Foo f1;
  Foo f2;
  Foo f3;
}
```

In the example above, the static class variable numFoos is shared between all three instances of the *Foo* class (*f1*, *f2* and *f3*) and keeps a count of the number of times that the Foo class has been instantiated.

4.3.5 Member Functions

Member functions can (and should) be used to interact with data contained within user defined types. User defined types provide flexibility in the "DIVIDE AND CONQUER[16]" scheme in program writing. In other words, one programmer can write a user defined type and guarantee an interface. Another programmer can write the main program with that expected interface. The two pieces are put together and compiled for usage. User defined types provide *encapsulation* defined in the Object Oriented Programming (OOP) paradigm.

Within classes, to protect the data members, the programmer can define functions to perform the operations on those data members. Member functions and functions are names used interchangeably in reference to classes. Function prototypes are declared within the class definition. These prototypes can take the form of non-class functions as well as class suitable prototypes. Functions can be declared and defined within the class definition. However, most functions can have very large definitions and make the class very unreadable. Therefore it is possible to define the function outside of the class definition using the scope resolution operator "**::**".

16 HTTP://EN.WIKIPEDIA.ORG/WIKI/DIVIDE%20AND%20CONQUER

This scope resolution operator allows a programmer to define the functions somewhere else. This can allow the programmer to provide a header file *.h* defining the class and a *.obj* file built from the compiled *.cpp* file which contains the function definitions. This can hide the implementation and prevent tampering. The user would have to define every function again to change the implementation. Functions within classes can access and modify (unless the function is constant) data members without declaring them, because the data members are already declared in the class.

Simple example:

file: Foo.h

```
// the header file named the same as the class helps locate classes within a
 project
// one class per header file makes it easier to keep the
// header file readable (some classes can become large)
// each programmer should determine what style works for them or what programming
 standards their
// teacher/professor/employer has

#ifndef FOO_H
#define FOO_H

class Foo{
public:
  Foo();                 // function called the default constructor
  Foo( int a, int b );    // function called the overloaded constructor
  int Manipulate( int g, int h );

private:
  int x;
  int y;
};

#endif
```

file: Foo.cpp

```
#include "Foo.h"

/* these constructors should really show use of initialization lists
Foo::Foo() : x(5), y(10)
{
}
Foo:Foo(int a, int b) : x(a), y(b)
{
}
*/
Foo::Foo(){
  x = 5;
  y = 10;
}
Foo::Foo( int a, int b ){
```

```
  x = a;
  y = b;
}

int Foo::Manipulate( int g, int h ){
  x = h + g*x;
  y = g + h*y;
}
```

Overloading

Member functions can be overloaded. This means that multiple member functions can exist with the same name on the same scope, but must have different signatures. A member function's signature is comprised of the member function's name and the type and order of the member function's parameters.

Due to name hiding, if a member in the derived class shares the same name with members of the base class, they will be hidden to the compiler. To make those members visible, one can use declarations to introduce them from base class scopes.

Constructors and other class member functions, except the Destructor, can be overloaded.

Constructors

A constructor is a special member function which is called whenever a new instance of a class is created. The compiler calls the constructor after the new object has been allocated in memory, and converts that "raw" memory into a proper, typed object. The constructor is declared much like a normal member function but it will share the name of the class and it has no return value.

Constructors are responsible for almost all of the run-time setup necessary for the class operation. Its main purpose becomes in general defining the data members upon object instantiation (when an object is declared), they can also have arguments, if the programmer so chooses. If a constructor has arguments, then they should also be added to the declaration of any other object of that class when using the **new** operator. Constructors can also be overloaded.

```
Foo myTest;                // essentially what happens is:  Foo myTest = Foo();
Foo myTest( 3, 54 );       // accessing the overloaded constructor
Foo myTest = Foo( 20, 45 ); // although a new object is created, there are some
 extra function calls involved
                           // with more complex classes, an assignment operator
```

```
 should
                          // be defined to ensure a proper copy (includes
''deep copy'')
                          // myTest would be constructed with the default
 constructor, and then the
                          // assignment operator copies the unnamed Foo( 20, 45
) object to myTest
```

using **new** with a constructor

```
Foo* myTest = new Foo();          // this defines a pointer to a dynamically
 allocated object
Foo* myTest = new Foo( 40, 34 );   // constructed with Foo( 40, 34 )
// be sure to use delete to avoid memory leaks
```

Note:
While there is no risk in using **new** to create an object, it is often best to avoid using memory allocation functions within objects' constructors. Specifically, using new to create an array of objects, each of which also uses new to allocate memory during its construction, often results in runtime errors. If a class or structure contains members which must be pointed at dynamically created objects, it is best to sequentially initialize these arrays of the parent object, rather than leaving the task to their constructors.

This is especially important when writing code with exceptions (in EXCEPTION HANDLING[a]), if an exception is thrown before a constructor is completed, the associated destructor will not be called for that object.

a Chapter 5.4 on page 517

A constructor can't delegate to another. It is also considered desirable to reduce the use of default arguments, if a maintainer has to write and maintain multiple constructors it can result in code duplication, which reduces maintainability because of the potential for introducing inconsistencies and even lead to code bloat.

Default Constructors

A default constructor is one which can be called with no arguments. Most commonly, a default constructor is declared without any parameters, but it is also possible for a constructor with parameters to be a default constructor if all of those parameters are given default values.

In order to create an array of objects of a class type, the class must have an accessible default constructor; C++ has no syntax to specify constructor arguments for array elements.

Overloaded Constructors

When an object of a class is instantiated, the class writer can provide various constructors each with a different purpose. A large class would have many data members, some of which may or may not be defined when an object is instantiated. Anyway, each project will vary, so a programmer should investigate various possibilities when providing constructors.

These are all constructors for a class myFoo.

```
myFoo(); // default constructor, the user has no control over initial values
         // overloaded constructors

myFoo( int a, int b=0 ); // allows construction with a certain 'a' value, but
 accepts 'b' as 0
                         // or allows the user to provide both 'a' and 'b' values
 // or

myFoo( int a, int b ); // overloaded constructor, the user must specify both
 values

class myFoo {
private:
  int Useful1;
  int Useful2;

public:
  myFoo(){                    // default constructor
           Useful1 = 5;
           Useful2 = 10;
          };

  myFoo( int a, int b = 0 ) { // two possible cases when invoked
         Useful1 = a;
         Useful2 = b;
  };

};

myFoo Find;           // default constructor, private member values Useful1 = 5,
 Useful2 = 10
myFoo Find( 8 );      // overloaded constructor case 1, private member values
 Useful1 = 8, Useful2 = 0
myFoo Find( 8, 256 ); // overloaded constructor case 2, private member values
 Useful1 = 8, Useful2 = 256
```

Constructor initialization lists

Constructor initialization lists (or member initialization list) are the only way to initialize data members and base classes with a non-default constructor. Constructors for the members are included between the argument list and the body of the constructor (separated from the argument list by a colon). Using the initialization lists is not only better in terms of efficiency but also the simplest way to guarantee that all initialization of data members are done before entering the body of constructors.

```
// Using the initialization list for _myComplexMember
MyClass::MyClass(int mySimpleMember, MyComplexClass myComplexMember)
: _myComplexMember(myComplexMember) // only 1 call, to the copy constructor
{
 _mySimpleMember=mySimpleMember; // uses 2 calls, one for the constructor of the
 mySimpleMember class
                                 // and a second for the assignment operator of
 the MyComplexClass class
}
```

This is more efficient than assigning value to the complex data member inside the body of the constructor because in that case the variable is initialized with its corresponding constructor.

Note that the arguments provided to the constructors of the members do not need to be arguments to the constructor of the class; they can also be constants. Therefore you can create a default constructor for a class containing a member with no default constructor.

Example:

```
MyClass::MyClass() : _myComplexMember(0) { }
```

It is useful to initialize your members in the constructor using this initialization lists. This makes it obvious for the reader that the constructor does not execute logic. The order the initialization is done should be the same as you defined your base-classes and members. Otherwise you can get warnings at compile-time. Once you start initializing your members make sure to keep all in the constructor(s) to avoid confusion and possible 0xbaadfood.

It is safe to use constructor parameters that are named like members.

Example:

```
class MyClass : public MyBaseClassA, public MyBaseClassB {
  private:
    int c;
    void *pointerMember;
  public:
```

```
    MyClass(int,int,int);
};
/*...*/
MyClass::MyClass(int a, int b, int c):
 MyBaseClassA(a)
,MyBaseClassB(b)
,c(c)
,pointerMember(NULL)
,referenceMember()
{
 //logic
}
```

Note that this technique was also possible for normal functions but it is now obsoleted and is classified as an error in such case.

Note:
It is a common misunderstanding that initialization of data members can be done within the body of constructors. All such kind of so-called "initialization" are actually assignments. The C++ standard defines that all initialization of data members are done before entering the body of constructors. This is the reason why certain types (const types and references) cannot be assigned to and must be initialized in the constructor initialization list.
One should also keep in mind that class members are initialized in the order they are declared, not the order they appear in the initializer list. One way of avoiding CHICKEN AND EGG PARADOXES[a] is to always add the members to the initializer list in the same order they're declared.

a HTTP://EN.WIKIPEDIA.ORG/WIKI/CHICKEN%20OR%20THE%20EGG

Destructors

Destructors like the Constructors are declared as any normal member functions but will share the same name as the Class, what distinguishes them is that the Destructor's name is preceded with a "~", it can not have arguments and can't be overloaded.

Destructors are called whenever an Object of the Class is destroyed. Destructors are crucial in avoiding resource leaks (by deallocating memory), and in implementing the RAII idiom. Resources which are allocated in a Constructor of a Class are usually released in the Destructor of that Class as to return the system to some known or stable state after the Class ceases to exist.

The Destructor is invoked when Objects are destroyed, after the function they were declared in returns, when the **delete** operator is used or when the program is over. If an object of a derived type is destructed, first the Destructor of the most derived object is executed. Then member objects and base class subjects are destructed recursively, in the reverse order their corresponding Constructors completed. As with structs the compiler implicitly-declares a Destructor as a inline public member of its class if the class doesn't have a user-declared Destructor.

The DYNAMIC TYPE[17] of the object will change from the most derived type as Destructors run, symmetrically to how it changes as Constructors execute. This affects the functions called by virtual calls during construction and destruction, and leads to the common (and reasonable) advice to avoid calling virtual functions of an object either directly or indirectly from its Constructors or Destructors.

`inline`

Sharing most of the concepts we have seen before on the introduction to INLINE FUNCTIONS[18], when dealing with member function those concepts are extended, with a few additional considerations.

If the member functions definition is included inside the declaration of the class, that function is by default made implicitly inline. Compiler options may override this behavior.

Calls to virtual functions cannot be inlined if the object's type is not known at compile-time, because we don't know which function to inline.

`static`

The **static** keyword can be used in four different ways:

- TO CREATE PERMANENT STORAGE FOR LOCAL VARIABLES IN A FUNCTION[19].
- TO SPECIFY INTERNAL LINKAGE[20].
- TO DECLARE MEMBER FUNCTIONS THAT ACT LIKE NON-MEMBER FUNCTIONS[21].

17 HTTP://EN.WIKIPEDIA.ORG/WIKI/DYNAMIC%20TYPE
18 Chapter 3.7 on page 229
19 Chapter 3.3.4 on page 156
20 Chapter 3.2.4 on page 119
21 Chapter 4.3.5 on page 415

- TO CREATE A SINGLE COPY OF A DATA MEMBER[22].

static member function

Member functions or variables declared static are shared between all instances of an object type. Meaning that only one copy of the member function or variable does exists for any object type.

member functions callable without an object

When used in a class function member, the function does not take an instantiation as an implicit `this` parameter, instead behaving like a free function. This means that static class functions can be called without creating instances of the class:

```
class Foo {
public:
  Foo() {
    ++numFoos;
    cout << "We have now created " << numFoos << " instances of the Foo class\n";
  }
  static int getNumFoos() {
    return numFoos;
  }
private:
  static int numFoos;
};

int Foo::numFoos = 0;  // allocate memory for numFoos, and initialize it

int main() {
  Foo f1;
  Foo f2;
  Foo f3;
  cout << "So far, we've made " << Foo::getNumFoos() << " instances of the Foo
 class\n";
}
```

Named constructors

Named constructors are a good example of using static member functions. *Named constructors* is the name given to functions used to create an object of a class without (directly) using its constructors. This might be used for the following:

1. To circumvent the restriction that constructors can be overloaded only if their signatures differ.
2. Making the class non-inheritable by making the constructors private.

22 Chapter 4.3.4 on page 406

3. Preventing stack allocation by making constructors private

Declare a static member function that uses a private constructor to create the object and return it. (It could also return a pointer or a reference but this complication seems useless, and turns this into the FACTORY PATTERN[23] rather than a conventional named constructor.)

Here's an example for a class that stores a temperature that can be specified in any of the different temperature scales.

```
class Temperature
{
    public:
        static Temperature Fahrenheit (double f);
        static Temperature Celsius (double c);
        static Temperature Kelvin (double k);
    private:
        Temperature (double temp);
        double _temp;
};

Temperature::Temperature (double temp):_temp (temp) {}

Temperature Temperature::Fahrenheit (double f)
{
    return Temperature ((f + 459.67) / 1.8);
}

Temperature Temperature::Celsius (double c)
{
    return Temperature (c + 273.15);
}

Temperature Temperature::Kelvin (double k)
{
    return Temperature (k);
}
```

const

This type of member function cannot modify the member variables of a class. It's a hint both to the programmer and the compiler that a given member function doesn't change the internal state of a class; however, any variables declared as `mutable` can still be modified.

Take for example:

23 Chapter 6.2 on page 541

```
class Foo
{
 public:
   int value() const
   {
     return m_value;
   }

   void setValue( int i )
   {
     m_value = i;
   }

 private:
   int m_value;
};
```

Here `value()` clearly does not change m_value and as such can and should be const. However `setValue()` does modify m_value and as such cannot be const.

Another subtlety often missed is a `const` member function cannot call a non-const member function (and the compiler will complain if you try). The `const` member function cannot change member variables and a non-const member functions can change member variables. Since we assume non-const member functions do change member variables, `const` member functions are assumed to never change member variables and can't call functions that do change member variables.

The following code example explains what `const` can do depending on where it is placed.

```
class Foo
{
public:
   /*
     * Modifies m_widget and the user
     * may modify the returned widget.
     */
   Widget *widget();

   /*
     * Does not modify m_widget but the
     * user may modify the returned widget.
     */
   Widget *widget() const;

   /*
     * Modifies m_widget, but the user
     * may not modify the returned widget.
     */
   const Widget *cWidget();

   /*
```

```
     * Does not modify m_widget and the user
     * may not modify the returned widget.
     */
   const Widget *cWidget() const;

private:
   Widget *m_widget;
};
```

Accessors and Modifiers (Setter/Getter)

What is an accessor?

An accessor is a member function that does not modify the state of an object. The accessor functions should be declared as CONST[24].

Getter is another common definition of an accessor due to the naming (`GetSize()`) of that type of member functions.

What is a modifier?

A modifier, also called a modifying function, is a member function that changes the value of at least one data member. In other words, an operation that modifies the state of an object. Modifiers are also known as 'mutators'.

Setter is another common definition of a modifier due to the naming (`SetSize( int a_Size )`) of that type of member functions.

Note:
These are commonly used reference labels (not defined on the standard language).

Dynamic polymorphism (Overrides)

So far, we have learned that we can add new data and functions to a class through inheritance. But what about if we want our derived class to inherit a method from the base class, but to have a different implementation for it? That is when we are talking about polymorphism, a fundamental concept in OOP programming.

24 Chapter 4.3.5 on page 409

As seen previously in the PROGRAMMING PARADIGMS SECTION[25], POLYMORPHISM[26] is subdivided in two concepts *static polymorphism* and *dynamic polymorphism*. This section concentrates on dynamic polymorphism, which applies in C++ when a derived class overrides a function declared in a base class.

We implement this concept redefining the method in the derived class. However, we need to have some considerations when we do this, so now we must introduce the concepts of dynamic binding, static binding and virtual methods.

Suppose that we have two classes, `A` and `B`. `B` derives from `A` and redefines the implementation of a method `c()` that resides in class `A`. Now suppose that we have an object `b` of class `B`. How should the instruction `b.c()` be interpreted?

If `b` is declared in the stack (not declared as a pointer or a reference) the compiler applies static binding, this means it interprets (at compile time) that we refer to the implementation of `c()` that resides in `B`.

However, if we declare `b` as a pointer or a reference of class `A`, the compiler could not know which method to call at compile time, because `b` can be of type `A` or `B`. If this is resolved at run time, the method that resides in `B` will be called. This is called dynamic binding. If this is resolved at compile time, the method that resides in A will be called. This is again, static binding.

Virtual member functions

The `virtual` member functions is relatively simple, but often misunderstood. The concept is an essential part of designing a class hierarchy in regards to sub-classing classes as it determines the behavior of overridden methods in certain contexts.

Virtual member functions are class member functions, that can be overridden in any class derived from the one where they were declared. The member function body is then replaced with a new set of implementation in the derived class.

Note:
When overriding virtual functions you can alter the private, protected or public state access state of the member function of the derived class.

25 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%2B%2B%20PROGRAMMING%2FPROGRAMMING%20PARADIGMS

26 Chapter 2.3.4 on page 21

By placing the keyword `virtual` before a method declaration we are indicating that when the compiler has to decide between applying static binding or dynamic binding it will apply dynamic binding. Otherwise, static binding will be applied.

Note:
While it is not required to use the virtual keyword in our subclass definitions (since if the base class function is virtual all subclass overrides of it will also be virtual) it is good style to do so when producing code for future reutilization (for use outside of the same project).

Again, this should be clearer with an example:

```
class Foo
{
public:
  void f()
  {
    std::cout << "Foo::f()" << std::endl;
  }
  virtual void g()
  {
    std::cout << "Foo::g()" << std::endl;
  }
};

class Bar : public Foo
{
public:
  void f()
  {
    std::cout << "Bar::f()" << std::endl;
  }
  virtual void g()
  {
    std::cout << "Bar::g()" << std::endl;
  }
};

int main()
{
  Foo foo;
  Bar bar;

  Foo *baz = &bar;
  Bar *quux = &bar;

  foo.f(); // "Foo::f()"
  foo.g(); // "Foo::g()"

  bar.f(); // "Bar::f()"
  bar.g(); // "Bar::g()"
```

```
  // So far everything we would expect...

  baz->f();  // "Foo::f()"
  baz->g();  // "Bar::g()"

  quux->f(); // "Bar::f()"
  quux->g(); // "Bar::g()"

  return 0;
}
```

Our first calls to `f()` and `g()` on the two objects are straightforward. However things get interesting with our baz pointer which is a pointer to the Foo type.

`f()` is not `virtual` and as such a call to `f()` will always invoke the implementation associated with the pointer type -- in this case the implementation from Foo.

Note:
Remember that OVERLOADING[a] and OVERRIDING[b] are distinct concepts.

a Chapter 4.3.5 on page 409
b Chapter 4.3.5 on page 418

Virtual function calls are computationally more expensive than regular function calls. Virtual functions use pointer indirection, invocation and will require a few extra instructions than normal member functions. They also require that the constructor of any class/structure containing virtual functions to initialize a table of pointers to its virtual member functions.

All this characteristics will signify a trade-off between performance and design. One should avoid preemptively declaring functions virtual without an existing structural need. Keep in mind that virtual functions that are only resolved at runtime cannot be inlined.

Note:
Some of the needs for using virtual functions can be addressed by using a class template. This will be covered when we introduce TEMPLATES[a].

a Chapter 5 on page 483

Pure virtual member function

There is one additional interesting possibility. Sometimes we don't want to provide an implementation of our function at all, but want to require people sub-

classing our class to provide an implementation on their own. This is the case for *pure* virtuals.

To indicate a pure `virtual` function instead of an implementation we simply add an "`= 0`" after the function declaration.

Again -- an example:

```
class Widget
{
public:
   virtual void paint() = 0;
};

class Button : public Widget
{
public:
   void paint() // is virtual because it is an override
   {
       // do some stuff to draw a button
   }
};
```

Because `paint()` is a pure `virtual` function in the Widget `class` we are required to provide an implementation in all concrete subclasses. If we don't the compiler will give us an error at build time.

This is helpful for providing interfaces -- things that we expect from all of the objects based on a certain hierarchy, but when we want to ignore the implementation details.

So why is this useful?

Let's take our example from above where we had a pure `virtual` for painting. There are a lot of cases where we want to be able to do things with widgets without worrying about what kind of widget it is. Painting is an easy example.

Imagine that we have something in our application that repaints widgets when they become active. It would just work with pointers to widgets -- i.e. `Widget *activeWidget() const` might be a possible function signature. So we might do something like:

```
Widget *w = window->activeWidget();
w->paint();
```

We want to actually call the appropriate paint member function for the "real" widget type -- not `Widget::paint()` (which is a "pure" `virtual` and will cause

the program to crash if called using virtual dispatch). By using a `virtual` function we insure that the member function implementation for our subclass -- `Button::paint()` in this case -- will be called.

Covariant return types

Covariant return types is the ability for a virtual function in a derived class to return a pointer or reference to an instance of itself if the version of the method in the base class does so. e.g.

```
class base
{
public:
  virtual base* create() const;
};

class derived : public base
{
public:
  virtual derived* create() const;
};
```

This allows casting to be avoided.

> **Note:**
> Some older compilers do not have support for covariant return types. Workarounds exist for such compilers.

virtual Constructors

There is a hierarchy of classes with base class `Foo`. Given an object `bar` belonging in the hierarchy, it is desired to be able to do the following:

1. Create an object `baz` of the same class as `bar` (say, class `Bar`) initialized using the default constructor of the class. The syntax normally used is:

   ```
   Bar* baz = bar.create();
   ```
2. Create an object `baz` of the same class as `bar` which is a copy of `bar`. The syntax normally used is:

   ```
   Bar* baz = bar.clone();
   ```

In the class `Foo`, the methods `Foo::create()` and `Foo::clone()` are declared as follows:

```
class Foo
{
    // ...

    public:
        // Virtual default constructor
        virtual Foo* create() const;

        // Virtual copy constructor
        virtual Foo* clone() const;
};
```

If `Foo` is to be used as an abstract class, the functions may be made pure virtual:

```
class Foo
{
   // ...

    public:
        virtual Foo* create() const = 0;
        virtual Foo* clone() const = 0;
};
```

In order to support the creation of a default-initialized object, and the creation of a copy object, each class `Bar` in the hierarchy must have public default and copy constructors. The virtual constructors of `Bar` are defined as follows:

```
class Bar : ... // Bar is a descendant of Foo
{
    // ...

    public:
    // Non-virtual default constructor
    Bar ();
    // Non-virtual copy constructor
    Bar (const Bar&);

    // Virtual default constructor, inline implementation
    Bar* create() const { return new Foo (); }
    // Virtual copy constructor, inline implementation
    Bar* clone() const { return new Foo (*this); }
};
```

The above code uses COVARIANT RETURN TYPES[27]. If your compiler doesn't support `Bar* Bar::create()`, use `Foo* Bar::create()` instead, and similarly for `clone()`.

While using these virtual constructors, you *must* manually deallocate the object created by calling **delete** `baz;`. This hassle could be avoided if a smart pointer

27 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23COVARIANT%20RETURN%20TYPES

(e.g. `std::auto_ptr<Foo>`) is used in the return type instead of the plain old `Foo*`.

Remember that whether or not `Foo` uses dynamically allocated memory, you *must* define the destructor **virtual** `~Foo ()` and make it `virtual` to take care of deallocation of objects using pointers to an ancestral type.

virtual Destructor

It is of special importance to remember to define a virtual destructor even if empty in any base class, since failing to do so will create problems with the default compiler generated destructor that will not be virtual.

A virtual destructor is not overridden when redefined in a derived class, the definitions to each destructor are cumulative and they start from the last derivate class toward the first base class.

Pure virtual Destructor

Every abstract class should contain the declaration of a pure virtual destructor.

Pure virtual destructors are a special case of pure virtual functions (meant to be overridden in a derived class). They must always be defined and that definition should always be empty.

```
class Interface {
public:
  virtual ~Interface() = 0; //declaration of a pure virtual destructor
};

Interface::~Interface(){} //pure virtual destructor definition (should always be
 empty)
```

28 29

Law of three

The "law of three" is not really a law, but rather a guideline: if a class needs an explicitly declared copy constructor, copy assignment operator, or destructor, then it usually needs all three.

28 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
29 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

There are exceptions to this rule (or, to look at it another way, refinements). For example, sometimes a destructor is explicitly declared just in order to make it `virtual`; in that case there's not necessarily a need to declare or implement the copy constructor and copy assignment operator.

Most classes should not declare any of the "big three" operations; classes that manage resources generally need all three.

4.3.6 Subsumption property

Subsumption is a property that all objects that reside in a class hierarchy must fulfill: an object of the base class can be substituted by an object that derives from it (directly or indirectly). All mammals are animals (they derive from them), and all cats are mammals. Therefore, because of the subsumption property we can "treat" any mammal as an animal and any cat as a mammal. This implies abstraction, because when we are "treating" a mammal as an animal, the only we should know about it is that it lives, it grows, etc, but nothing related to mammals.

This property is applied in C++, whenever we are using pointers or references to objects that reside in a class hierarchy. In other words, a pointer of class animal can point to an object of class animal, mammal or cat.

Let's continue with our example:

```
//needs to be corrected
 enum AnimalType {
      Herbivore,
      Carnivore,
      Omnivore,
 };

 class Animal {
       public:
              AnimalType Type;
              bool bIsAlive;
              int iNumberOfChildren;
 };

 class Mammal : public Animal{
       public:
              int iNumberOfTeats;
 };

 class Cat : public Mammal{
       public:
             bool bLikesFish;  // probably true
 };
```

```
int main() {
    Animal* pA1 = new Animal;
    Animal* pA2 = new Mammal;
    Animal* pA3 = new Cat;
    Mammal* pM  = new Cat;

    pA2->bIsAlive = True;     // Correct
    pA2->Type = Herbivore;    // Correct
    pM->iNumberOfTeats = 2;   // Correct

    pA2->iNumberOfTeats = 6;  // Incorrect
    pA3->bLikesFish = True;   // Incorrect

    Cat* pC = (Cat*)pA3;      // Downcast, correct (but very poor practice, see
later)
    pC->bLikesFish = False;   // Correct (although it is a very awkward cat)
  }
```

In the last lines of the example there is cast of a pointer to *Animal*, to a pointer to *Cat*. This is called "Downcast". Downcasts are useful and should be used, but first we must ensure that the object we are casting is really of the type we are casting to it. Downcasting a base class to an unrelated class is an error. To resolve this issue, the casting operators `dynamic_cast`, or `static_cast<>` should be used. These correctly cast an object from one class to another, and will throw an exception if the class types are not related. eg. If you try:

```
Cat* pC = new Cat;

motorbike* pM = dynamic_cast<motorbike*>(pC);
```

Then, the app will throw an exception, as a cat is not a motorbike. Static_cast is very similar, only it will perform the type checking at compile time. If you have an object where you are not sure of its type then you should use `dynamic_cast`, and be prepared to handle errors when casting. If you are downcasting objects where you know the types, then you should use `static_cast`. Do not use old-style C casts as these will simply give you an access violation if the types cast are unrelated.

4.3.7 Local classes

A **local class** is any class that is defined inside a specific statement block, in a LOCAL SCOPE[30], for instance inside a function. This is done like defining any other class, but *local classes* can not however access non-static local variables or

30 Chapter 3.1.9 on page 78

be used to define STATIC DATA MEMBERS[31]. These type of classes are useful especially in template functions, as we will see later.

```
void MyFunction()
{
   class LocalClass
   {
   // ... members definitions ...
   };

   // ... any code that needs the class ...

}
```

4.3.8 User defined automatic type conversion

We already covered AUTOMATIC TYPE CONVERSIONS[32] (implicit conversion) and mentioned that some can be user-defined.

A user-defined conversion from a class to another class can be done by providing a constructor in the target class that takes the source class as an argument, `Target(const Source& a_Class)` or by providing the target class with a conversion operator, as `operator Source()`.

4.3.9 Ensuring objects of a class are never copied

This is required e.g. to prevent memory-related problems that would result in case the default copy-constructor or the default assignment operator is unintentionally applied to a class `C` which uses dynamically allocated memory, where a copy-constructor and an assignment operator are probably an overkill as they won't be used frequently.

Some style guidelines suggest making all classes non-copyable by default, and only enabling copying if it makes sense. Other (bad) guidelines say that you should always explicitly write the copy constructor and copy assignment operators; that's actually a bad idea, as it adds to the maintenance effort, adds to the work to read a class, is more likely to introduce errors than using the implicitly declared ones, and doesn't make sense for most object types. A sensible guideline is to *think* about whether copying makes sense for a type; if it does, then first prefer to arrange

31 Chapter 4.3.4 on page 406
32 Chapter 3.5.1 on page 205

that the compiler-generated copy operations will do the right thing (e.g., by holding all resources via resource management classes rather than via raw pointers or handles), and if that's not reasonable then obey the LAW OF THREE[33]. If copying doesn't make sense, you can disallow it in either of two idiomatic ways as shown below.

Just declare the copy-constructor and assignment operator, and make them `private`. Do not define them. As they are not `protected` or `public`, they are inaccessible outside the class. Using them within the class would give a linker error since they are not defined.

```
class C
{
  ...

  private:
    // Not defined anywhere
    C (const C&);
    C& operator= (const C&);
};
```

Remember that if the class uses dynamically allocated memory for data members, you *must* define the memory release procedures in destructor `~C ()` to release the allocated memory.

A class which only declares these two functions can be used as a private base class, so that all classes which privately inherits such a class will disallow copying.

Note:
A part of the BOOST[a] library, the utility class `boost:noncopyable` performs a similar function, easier to use but with added costs due to the required derivation.

a Chapter 6.4.2 on page 588

4.3.10 Container class

A class that is used to hold objects in memory or external storage is often called a *container class*. A container class acts as a generic holder and has a predefined behavior and a well-known interface. It is also a supporting class whose purpose is to hide the topology used for maintaining the list of objects in memory. When

33 Chapter 4.3.5 on page 409

it contains a group of mixed objects, the container is called a heterogeneous container; when the container is holding a group of objects that are all the same, the container is called a homogeneous container.

4.3.11 Interface class

4.3.12 Singleton class

A SINGLETON[34] class is a class that can only be instantiated once (similar to the use of static variables or functions). It is one of the possible implementations of a CREATIONAL PATTERN[35], which is fully covered in the DESIGN PATTERNS SECTION[36] of the book.

37

4.3.13 Abstract Classes

An abstract class is, conceptually, a class that cannot be instantiated and is usually implemented as a class that has one or more pure virtual (abstract) functions.

A pure virtual function is one which **must be overridden** by any concrete (i.e., non-abstract) derived class. This is indicated in the declaration with the syntax**" = 0"** in the member function's declaration.

Example

```
class AbstractClass {
public:
  virtual void AbstractMemberFunction() = 0; //pure virtual function makes this
 class Abstract class
  virtual void NonAbstractMemberFunction1(); //virtual function

  void NonAbstractMemberFunction2();
};
```

In general an abstract class is used to define an implementation and is intended to be inherited from by concrete classes. It's a way of forcing a contract between the

34 Chapter 6.3 on page 542
35 Chapter 6.3 on page 542
36 Chapter 6.2 on page 541
37 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

class designer and the users of that class. If we wish to create a concrete class (a class that can be instantiated) from an abstract class we must declare and **define** a matching member function for each abstract member function of the base class. Otherwise we will create a new abstract class (this could be useful sometimes).

Sometimes we use the phrase "pure abstract class," meaning a class that exclusively has pure virtual functions (and no data). The concept of interface is mapped to pure abstract classes in C++, as there is no construction "interface" in C++ the same way that there is in Java.

Example

```
class Vehicle {
public:
    explicit
    Vehicle( int topSpeed )
    : m_topSpeed( topSpeed )
    {}
    int TopSpeed() const {
       return m_topSpeed;
    }

    virtual void Save( std::ostream& ) const = 0;

private:
    int m_topSpeed;
};

class WheeledLandVehicle : public Vehicle {
public:
    WheeledLandVehicle( int topSpeed, int numberOfWheels )
    : Vehicle( topSpeed ), m_numberOfWheels( numberOfWheels )
    {}
    int NumberOfWheels() const {
      return m_numberOfWheels;
    }

    void Save( std::ostream& ) const; // is implicitly virtual

private:
    int m_numberOfWheels;
};

class TrackedLandVehicle : public Vehicle {
public:
   int TrackedLandVehicle ( int topSpeed, int numberOfTracks )
   : Vehicle( topSpeed), m_numberOfTracks ( numberOfTracks )
   {}
   int NumberOfTracks() const {
      return m_numberOfTracks;
   }
   void Save( std::ostream& ) const; // is implicitly virtual
```

```
  private:
    int m_numberOfTracks;
  };
```

In this example the Vehicle is an abstract base class as it has an abstract member function. It is not a pure abstract class as it also has data and concrete member functions. The class WheeledLandVehicle is derived from the base class. It also holds data which is common to all wheeled land vehicles, namely the number of wheels. The class TrackedLandVehicle is another variation of the Vehicle class.

This is something of a contrived example but it does show how that you can share implementation details among a hierarchy of classes. Each class further refines a concept. This is not always the best way to implement an interface but in some cases it works very well. As a guideline, for ease of maintenance and understanding you should try to limit the inheritance to no more than 3 levels. Often the best set of classes to use is a pure virtual abstract base class to define a common interface. Then use an abstract class to further refine an implementation for a set of concrete classes and lastly define the set of concrete classes.

An **abstract class** is a class that is designed to be specifically used as a base class. An abstract class contains at least one pure virtual function. You declare a pure virtual function by using a pure specifier (= 0) in the declaration of a virtual member function in the class declaration.

The following is an example of an abstract class:

```
class AB {
public:
  virtual void f() = 0;
};
```

Function AB::f is a pure virtual function. A function declaration cannot have both a pure specifier and a definition.

Abstract class cannot be used as a parameter type, a function return type, or the type of an explicit conversion, and not to declare an object of an abstract class. It can be used to declare pointers and references to an abstract class.

Pure Abstract Classes

An abstract class is one in which there is a declaration but no definition for a member function. The way this concept is expressed in C++ is to have the member function declaration assigned to zero.

Example

```
class PureAbstractClass
{
public:
  virtual void AbstractMemberFunction() = 0;
};
```

A pure Abstract class has only abstract member functions and no data or concrete member functions. In general, a pure abstract class is used to define an interface and is intended to be inherited by concrete classes. It's a way of forcing a contract between the class designer and the users of that class. The users of this class must declare a matching member function for the class to compile.

Example of usage for a pure Abstract Class

```
class DrawableObject
{
 public:
   virtual void Draw(GraphicalDrawingBoard&) const = 0; //draw to
GraphicalDrawingBoard
};

class Triangle : public DrawableObject
{
public:
  void Draw(GraphicalDrawingBoard&) const; //draw a triangle
};

class Rectangle : public DrawableObject
{
public:
  void Draw(GraphicalDrawingBoard&) const; //draw a rectangle
};

class Circle : public DrawableObject
{
public:
  void Draw(GraphicalDrawingBoard&) const; //draw a circle
};

typedef std::list<DrawableObject*> DrawableList_t;
```

```
DrawableList_t drawableList;
GraphicalDrawingBoard gdrawb;

drawableList.pushback(new Triangle());
drawableList.pushback(new Rectangle());
drawableList.pushback(new Circle());

for(DrawableList_t::const_iterator iter = drawableList.begin(),
   endIter = drawableList.end();
   iter != endIter;
   ++iter)
{
  DrawableObject *object = *iter;
  object->Draw(gdrawb);
}
```

Note that this is a bit of a contrived example and that the drawable objects are not fully defined (no constructors or data) but it should give you the general idea of the power of defining an interface. Once the objects are constructed, the code that calls the interface does not know any of the implementation details of the called objects, only that of the interface. The object *GraphicalDrawingBoard* is a placeholder meant to represent the thing onto which the object will be drawn, i.e. the video memory, drawing buffer, printer.

Note that there is a great temptation to add concrete member functions and data to pure abstract base classes. This must be resisted, in general it is a sign that the interface is not well factored. Data and concrete member functions tend to imply a particular implementation and as such can inherit from the interface but should not be that interface. Instead if there is some commonality between concrete classes, creation of abstract class which inherits its interface from the pure abstract class and defines the common data and member functions of the concrete classes works well. Some care should be taken to decide whether inheritance or aggregation should be used. Too many layers of inheritance can make the maintenance and usage of a class difficult. Generally, the maximum accepted layers of inheritance is about 3, above that and refactoring of the classes is generally called for. A general test is the "is a" vs "has a", as in a Square is a Rectangle, but a Square has a set of sides.

38

4.3.14 What is a "nice" class?

A "nice" class takes into consideration the use of the following functions:

38 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

1. The copy constructor.

2. The assignment operator.

3. The equality operator.

4. The inequality operator.

Class Declaration

```
class Nice
{
public:
    Nice(const Nice &Copy);
    Nice &operator= (const Nice &Copy);
    bool operator== (const Nice &param) const;
    bool operator!= (const Nice &param) const;
};
```

Description

A "nice" class could also be called a container safe class. Many containers such as those in the STANDARD TEMPLATE LIBRARY[39] (STL), that we'll see later, use copy construction and the assignment operator when interacting with the objects of your class. The assignment operator and copy constructor only need to be declared and defined if the default behavior, which is a member-wise (not binary) copy, is undesirable or insufficient to properly copy/construct your object.

A general rule of thumb is that if the default, member-wise copy operations do not work for your objects then you should define a suitable copy constructor and assignment operator. They are both needed if either is defined.

39 Chapter 5.1.5 on page 499

4.4 Copy Constructor

The purpose of the copy constructor is to allow the programmer to perform the same instructions as the assignment operator with the special case of knowing that the caller is initializing/constructing rather than an copying.

It is also good practice to use the explicit keyword when using a copy constructor to prevent unintended implicit type conversion.

Example

```
class Nice
{
public:
  explicit Nice(int _a) : a(_a)
  {
    return;
  }
private:
  int a;
};

class NotNice
{
public:
  NotNice(int _a) :  a(_a)
  {
    return;
  }
private:
  int a;
};

int main()
{
  Nice proper = Nice(10); //this is ok
  Nice notproper = 10; //this will result in an error
  NotNice eg = 10; //this WILL compile, you may not have intended this conversion
  return 0;
}
```

4.5 Equality Operator

The equality operator says, "Is this object equal to that object?". What constitutes equal is up to the programmer. This is a requirement if you ever want to use the equality operator with objects of your class.

However, in most applications (e.g. mathematics), it is usually the case that coding the inequality is easier than coding the equality. In which case the following code can be written for the equality.

```
inline bool Nice::operator== (const Nice& param) const
{
    return !(*this != param);
}
```

4.6 Inequality Operator

The inequality operator says, "Is this object not equal to that object?". What constitutes not equal is up to the programmer. This is a requirement if you ever want to use the inequality operator with objects of your class.

However, in some applications, coding the equality is easier than coding the inequality. In which case the following code can be written for the inequality.

```
inline bool Nice::operator!= (const Nice& param) const
{
    return !(*this == param);
}
```

If the statement about the (in)equality operators having different efficiency (whatever kind) seems complete nonsense to you, consider that *typically*, all object attributes must match for two objects to be considered equal.
Typically, only one object attribute must differ for two objects to be considered unequal. For equality and inequality operators, that doesn't mean one is faster than the other.

Note, however, that using both the above equality and inequality functions as defined will result in an infinite recursive loop and care must be taken to use only one or the other. Also, there are some situations where neither applies and therefore neither of the above can be used.

Given two objects A and B (with class attributes x and y), an equality operator could be written as

```
if (A.x != B.x) return false;
if (A.y != B.y) return false;
return true;
```

while an inequality operator could be written as

```
if (A.x != B.x) return true;
if (A.y != B.y) return true;
return false;
```

So yes, the equality operator can certainly be written *...!(a!=b)...*, but it isn't any faster. In fact, there's the additional overhead of a method call and a negation operation.

So the question becomes, is a little execution overhead worth the smaller code and improved maintainability? There is no simple answer to this it all depend on how the programmer is using them. If your class is composed of, say, an array of 1 billion elements, the overhead is negligible.

40

4.7 Operator overloading

Operator overloading (less commonly known as AD-HOC[41] POLYMORPHISM[42]) is a specific case of POLYMORPHISM[43] (part of the OO nature of the language) in which some or all operators like +, = or == are treated as polymorphic functions and as such have different behaviors depending on the types of its arguments. Operator overloading is usually only SYNTACTIC SUGAR[44]. It can easily be emulated using function calls.

Consider this operation:

```
add (a, multiply (b,c))
```

Using operator overloading permits a more concise way of writing it, like this:

```
a + b × c
```

(Assuming the × operator has higher PRECEDENCE[45] than +.)

40 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
41 HTTP://EN.WIKIPEDIA.ORG/WIKI/AD-HOC
42 HTTP://EN.WIKIPEDIA.ORG/WIKI/TYPE%20POLYMORPHISM
43 HTTP://EN.WIKIPEDIA.ORG/WIKI/POLYMORPHISM%20%28COMPUTER%20SCIENCE%29
44 HTTP://EN.WIKIPEDIA.ORG/WIKI/SYNTACTIC%20SUGAR
45 HTTP://EN.WIKIPEDIA.ORG/WIKI/PRECEDENCE

Operator overloading can provide more than an aesthetic benefit, since the language allows operators to be invoked implicitly in some circumstances. Problems, and critics, to the use of operator overloading arise because it allows programmers to give operators completely free functionality, without an imposition of coherency that permits to consistently satisfy user/reader expectations, usage of the << operator is an example of this problem.

```
// The expression
a << 1;
```

Will return twice the value of `a` if `a` is an integer variable, but if `a` is an output stream instead this will write "1" to it. Because operator overloading allows the programmer to change the usual semantics of an operator, it is usually considered good practice to use operator overloading with care.

To overload an operator is to provide it with a new meaning for user-defined types. This is done in the same fashion as defining a function. The basic syntax follows (where @ represents a valid operator):

```
return_type operator@(argument_list)
{
    // ... definition
}
```

Not all operators may be overloaded, new operators cannot be created, and the precedence, associativity or arity of operators cannot be changed (for example ! cannot be overloaded as a binary operator). Most operators may be overloaded as either a member function or non-member function, some, however, must be defined as member functions. Operators should only be overloaded where their use would be natural and unambiguous, and they should perform as expected. For example, overloading + to add two complex numbers is a good use, whereas overloading * to push an object onto a vector would not be considered good style.

Note:
Operator overloading should only be utilized when the meaning of the overloaded operator's operation is unambiguous and practical for the underlying type and where it would offer a significant notational brevity over appropriately named function calls.

A simple Message Header

```
// sample of Operator Overloading
```

```
#include <string>

class PlMessageHeader
{
    std::string m_ThreadSender;
    std::string m_ThreadReceiver;

    //return true if the messages are equal, false otherwise
    inline bool operator == (const PlMessageHeader &b) const
    {
        return ( (b.m_ThreadSender==m_ThreadSender) &&
                (b.m_ThreadReceiver==m_ThreadReceiver) );
    }

    //return true if the message is for name
    inline bool isFor (const std::string &name) const
    {
        return (m_ThreadReceiver==name);
    }

    //return true if the message is for name
    inline bool isFor (const char *name) const
    {
        return (m_ThreadReceiver==name);// since name type is std::string, it
 becomes unsafe if name == NULL
    }
};
```

Note:
The use of the `inline` keyword in the example above is technically redundant, as functions defined within a class definition like this are implicitly inline.

4.7.1 Operators as member functions

Aside from the operators which must be members, operators may be overloaded as member or non-member functions. The choice of whether or not to overload as a member is up to the programmer. Operators are generally overloaded as members when they:

1. change the left-hand operand, or
2. require direct access to the non-public parts of an object.

When an operator is defined as a member, the number of explicit parameters is reduced by one, as the calling object is implicitly supplied as an operand. Thus, binary operators take one explicit parameter and unary operators none. In the case of binary operators, the left hand operand is the calling object, and no type

COERCION[46] will be done upon it. This is in contrast to non-member operators, where the left hand operand may be coerced.

```
// binary operator as member function Vector2D Vector2D::operator+(const
Vector2D right)const {...}

// binary operator as non-member function Vector2D operator+(const
Vector2D left, const Vector2D right) {...}

// binary operator as non-member function with 2 arguments  friend
Vector2D operator+(const Vector2D left, const Vector2D right) {...}

// unary operator as member function Vector2D Vector2D::operator-()const
{...}

// unary operator as non-member function Vector2D operator-(const Vector2D
vec) {...}
```

4.7.2 Overloadable operators

Arithmetic operators

- **+** (addition)
- **-** (subtraction)
- ***** (multiplication)
- **/** (division)
- **%** (modulus)

As binary operators, these involve two arguments which do not have to be the same type. These operators may be defined as member or non-member functions. An example illustrating overloading for the addition of a 2D mathematical vector type follows.

```
Vector2D Vector2D::operator+(const Vector2D& right)
{
    Vector2D result;
    result.set_x(x() + right.x());
    result.set_y(y() + right.y());
    return result;
}
```

46 Chapter 3.3 on page 121

It is good style to only overload these operators to perform their customary arithmetic operation. Because operator has been overloaded as member function, it can access to private fields.

Bitwise operators

- ^ (XOR)
- | (OR)
- **&** (AND)
- ~ (complement)
- **<<** (shift left, insertion to stream)
- **>>** (shift right, extraction from stream)

All of the bitwise operators are binary, excepting complement, which is unary. It should be noted that these operators have a lower precedence than the arithmetic operators, so if ^ were to be overloaded for exponentiation, x ^ y + z may not work as intended. Of special mention are the shift operators, << and >>. These have been overloaded in the standard library for interaction with streams. When overloading these operators to work with streams the rules below should be followed:

1. overload << and >> as friends (so that it can access the private variables with the stream be passed in by references
2. (input/output modifies the stream, and copying is not allowed)
3. the operator should return a reference to the stream it receives (to allow chaining, cout << 3 << 4 << 5)

An example using a 2D vector

```
friend ostream& operator<<(ostream& out, const Vector2D& vec) // output
{
    out << "(" << vec.x() << ", " << vec.y() << ")";
    return out;
}

friend istream& operator>>(istream& in, Vector2D& vec) // input
{
    double x, y;
    in >> x >> y;
    vec.set_x(x);
    vec.set_y(y);
    return in;
}
```

Assignment operator

The assignment operator, **=**, **must be a member function**, and is given default behavior for user-defined classes by the compiler, performing an assignment of every member using its assignment operator. This behavior is generally acceptable for simple classes which only contain variables. However, where a class contains references or pointers to outside resources, the assignment operator should be overloaded (as general rule, whenever a destructor and copy constructor are needed so is the assignment operator), otherwise, for example, two strings would share the same buffer and changing one would change the other.

In this case, an assignment operator should perform two duties:

1. clean up the old contents of the object
2. copy the resources of the other object

For classes which contain raw pointers, before doing the assignment, the assignment operator should check for self-assignment, which generally will not work (as when the old contents of the object are erased, they cannot be copied to refill the object). Self assignment is generally a sign of a coding error, and thus for classes without raw pointers, this check is often omitted, as while the action is wasteful of cpu cycles, it has no other effect on the code.

Example

```
class BuggyRawPointer { // example of super-common mistake
    T *m_ptr;
    public:
    BuggyRawPointer(T *ptr) : m_ptr(ptr) {}
    BuggyRawPointer& operator=(BuggyRawPointer const &rhs) {
        delete m_ptr; // free resource; // Problem here!
        m_ptr = 0;
        m_ptr = rhs.m_ptr;
        return *this;
    };
};

BuggyRawPointer x(new T);
x = x; // We might expect this to keep x the same. This sets x.m_ptr == 0. Oops!

// The above problem can be fixed like so:
class WithRawPointer2 {
    T *m_ptr;
    public:
    WithRawPointer2(T *ptr) : m_ptr(ptr) {}
    WithRawPointer2& operator=(WithRawPointer2 const &rhs) {
        if (this != &rhs) {
            delete m_ptr; // free resource;
            m_ptr = 0;
```

```
            m_ptr = rhs.m_ptr;
        }
        return *this;
    };
};

WithRawPointer2 x2(new T);
x2 = x2; // x2.m_ptr unchanged.
```

Another common use of overloading the assignment operator is to declare the overload in the private part of the class and not define it. Thus any code which attempts to do an assignment will fail on two accounts, first by referencing a private member function and second fail to link by not having a valid definition. This is done for classes where copying is to be prevented, and generally done with the addition of a privately declared copy constructor

Example

```
class DoNotCopyOrAssign {
    public:
        DoNotCopyOrAssign() {};
    private:
        DoNotCopyOrAssign(DoNotCopyOrAssign const&);
        DoNotCopyOrAssign &operator=(DoNotCopyOrAssign const &);
};

class MyClass : public DoNotCopyOrAssign {
    public:
        MyClass();
};

MyClass x, y;
x = y; // Fails to compile due to private assignment operator;
MyClass z(x); // Fails to compile due to private copy constructor.
```

Relational operators

- **==** (equality)
- **!=** (inequality)
- **>** (greater-than)
- **<** (less-than)
- **>=** (greater-than-or-equal-to)
- **<=** (less-than-or-equal-to)

All relational operators are binary, and should return either true or false. Generally, all six operators can be based off a comparison function, or each other, although

this is never done automatically (e.g. overloading > will not automatically overload < to give the opposite). There are, however, some templates defined in the header <utility>; if this header is included, then it suffices to just overload operator== and operator<, and the other operators will be provided by the STL.

Logical operators

- **!** (NOT)
- **&&** (AND)
- || (OR)

The ! operator is unary, && and || are binary. It should be noted that in normal use, && and || have "short-circuit" behavior, where the right operand may not be evaluated, depending on the left operand. When overloaded, these operators get function call precedence, and this short circuit behavior is lost. It is best to leave these operators alone.

Example

```
bool Function1();
bool Function2();

Function1() && Function2();
```

If the result of Function1() is false, then Function2() is not called.

```
MyBool Function3();
MyBool Function4();

bool operator&&(MyBool const &, MyBool const &);

Function3() && Function4()
```

Both Function3() and Function4() will be called no matter what the result of the call is to Function3() This is a waste of CPU processing, and worse, it could have surprising unintended consequences compared to the expected "short-circuit" behavior of the default operators. Consider:

```
extern MyObject * ObjectPointer;
bool Function1() { return ObjectPointer != null; }
bool Function2() { return ObjectPointer->MyMethod(); }
MyBool Function3() { return ObjectPointer != null; }
MyBool Function4() { return ObjectPointer->MyMethod(); }

bool operator&&(MyBool const &, MyBool const &);
```

```
Function1() && Function2(); // Does not execute Function2() when pointer is null
Function3() && Function4(); // Executes Function4() when pointer is null
```

Compound assignment operators

- **+=** (addition-assignment)
- **-=** (subtraction-assignment)
- ***=** (multiplication-assignment)
- **/=** (division-assignment)
- **%=** (modulus-assignment)
- **&=** (AND-assignment)
- **|=** (OR-assignment)
- **^=** (XOR-assignment)
- **>>=** (shift-right-assignment)
- **<<=** (shift-left-assignment)

Compound assignment operators should be overloaded as member functions, as they change the left-hand operand. Like all other operators (except basic assignment), compound assignment operators must be explicitly defined, they will not be automatically (e.g. overloading = and + will not automatically overload +=). A compound assignment operator should work as expected: A @= B should be equivalent to A = A @ B. An example of += for a two-dimensional mathematical vector type follows.

```
Vector2D& Vector2D::operator+=(const Vector2D& right)
{
    this->x += right.x;
    this->y += right.y;
    return *this;
}
```

Increment and decrement operators

- **++** (increment)
- **--** (decrement)

Increment and decrement have two forms, prefix (++i) and postfix (i++). To differentiate, the postfix version takes a dummy integer. Increment and decrement operators are most often member functions, as they generally need access to the private member data in the class. The prefix version in general should return a reference to the changed object. The postfix version should just return a copy of the original value. In a perfect world, A += 1, A = A + 1, A++, ++A should all leave A with the same value.

Example

```
SomeValue SomeValue::operator++() // prefix { ++data; return *this; }

SomeValue SomeValue::operator++(int unused) // postfix { SomeValue result =
*this; ++data; return result; }
```

Often one operator is defined in terms of the other for ease in maintenance, especially if the function call is complex.

```
SomeValue SomeValue::operator++(int unused) // postfix
{
    SomeValue result = *this;
    ++(*this); // call SomeValue::operator++()
    return result;
}
```

Subscript operator

The subscript operator, **[ ]**, is a binary operator which **must be a member function** (hence it takes only one explicit parameter, the index). The subscript operator is not limited to taking an integral index. For instance, the index for the subscript operator for the std::map template is the same as the type of the key, so it may be a string etc. The subscript operator is generally overloaded twice; as a non-constant function (for when elements are altered), and as a constant function (for when elements are only accessed).

Function call operator

The function call operator, **( )**, is generally overloaded to create objects which behave like functions, or for classes that have a primary operation. The function call operator must be a member function, but has no other restrictions - it may be overloaded with any number of parameters of any type, and may return any type. A class may also have several definitions for the function call operator.

Address of, Reference, and Pointer operators

These three operators, operator&(), operator*() and operator->() can be overloaded. In general these operators are only overloaded for smart pointers, or classes which attempt to mimic the behavior of a raw pointer. The pointer operator, operator->() has the additional requirement that the result of the call to that

operator, must return a pointer, or a class with an overloaded operator->(). In general A == *&A should be true.

Example

```
class T {
    public:
        const memberFunction() const;
};

// forward declaration
class DullSmartReference;

class DullSmartPointer {
    private:
        T *m_ptr;
    public:
        DullSmartPointer(T *rhs) : m_ptr(rhs) {};
        DullSmartReference operator*() const {
            return DullSmartReference(*m_ptr);
        }
        T *operator->() const {
            return m_ptr;
        }
};

class DullSmartReference {
    private:
        T *m_ptr;
    public:
        DullSmartReference (T &rhs) : m_ptr(&rhs) {}
        DullSmartPointer operator&() const {
            return DullSmartPointer(m_ptr);
        }
        // conversion operator
        operator T() { return *m_ptr; }
};

DullSmartPointer dsp(new T);
dsp->memberFunction(); // calls T::memberFunction

T t;
DullSmartReference dsr(t);
dsp = &dsr;
t = dsr; // calls the conversion operator
```

These are extremely simplified examples designed to show how the operators can be overloaded and not the full details of a SmartPointer or SmartReference class. In general you won't want to overload all three of these operators in the same class.

Comma operator

The comma operator,() **,** can be overloaded. The language comma operator has left to right precedence, the operator,() has function call precedence, so be aware that overloading the comma operator has many pitfalls.

Example

```
MyClass operator,(MyClass const &, MyClass const &);

MyClass Function1();
MyClass Function2();

MyClass x = Function1(), Function2();
```

For non overloaded comma operator, the order of execution will be Function1(), Function2(); With the overloaded comma operator, the compiler can call either Function1(), or Function2() first.

Member access operators

The two member access operators, operator->() and operator->*() can be overloaded. The most common use of overloading these operators is with defining expression template classes, which is not a common programming technique. Clearly by overloading these operators you can create some very unmaintainable code so overload these operators only with great care.

When the -> operator is applied to a pointer value of type (T *), the language dereferences the pointer and applies the . member access operator (so x->m is equivalent to (*x).m). However, when the -> operator is applied to a class instance, it is called as a unary postfix operator; it is expected to return a value to which the -> operator can again be applied. Typically, this will be a value of type (T *), as in the example under ADDRESS OF, REFERENCE, AND POINTER OPERATORS[47] above, but can also be a class instance with operator->() defined; the language will call operator->() as many times as necessary until it arrives at a value of type (T *).

Memory management operators

47 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23ADDRESS%20OF.2C%20REFERENCE.2C%20AND%20POINTER%20OPERATORS

- **new** (allocate memory for object)
- **new[ ]** (allocate memory for array)
- **delete** (deallocate memory for object)
- **delete[ ]** (deallocate memory for array)

The memory management operators can be overloaded to customize allocation and deallocation (e.g. to insert pertinent memory headers). They should behave as expected, **new** should return a pointer to a newly allocated object on the heap, **delete** should deallocate memory, ignoring a NULL argument. To overload **new**, several rules must be followed:

- **new** must be a member function
- the return type must be *void**
- the first explicit parameter must be a *size_t* value

To overload **delete** there are also conditions:

- **delete** must be a member function (and cannot be virtual)
- the return type must be *void*
- there are only two forms available for the parameter list, and only one of the forms may appear in a class:
  - *void**
  - *void*, size_t*

Conversion operators
Conversion operators enable objects of a class to be either implicitly (coercion) or explicitly (casting) converted to another type. Conversion operators must be member functions, and should not change the object which is being converted, so should be flagged as constant functions. The basic syntax of a conversion operator declaration, and declaration for an int-conversion operator follows.

```
operator ''type''() const; // const is not necessary, but is good style
operator int() const;
```

Notice that the function is declared without a return-type, which can easily be inferred from the type of conversion. Including the return type in the function header for a conversion operator is a syntax error.

```
double operator double() const; // error - return type included
```

4.7.3 Operators which cannot be overloaded

- **?:** (conditional)
- **.** (member selection)

- **.*** (member selection with pointer-to-member)
- **::** (scope resolution)
- `sizeof` (object size information)
- **typeid** (object type information)

To understand the reasons why the language doesn't permit these operators to be overloaded, read "Why can't I overload dot, ::, `sizeof`, etc.?" at the Bjarne Stroustrup's C++ Style and Technique FAQ (HTTP://WWW.RESEARCH.ATT.COM/~BS/BS_FAQ2.HTML#OVERLOAD-DOT[48]).

[49]

4.8 I/O

Also commonly referenced as the C++ I/O of the C++ STANDARD LIBRARY[50], since the library also includes the C Standard library and its I/O implementation, as seen before in the STANDARD C I/O SECTION[51].

Input and *output* are essential for any computer software, as these are the only means by which the program can communicate with the user. The simplest form of input/output is pure textual, i.e. the application displays in console form, using simple ASCII characters to prompt the user for inputs, which are supplied using the keyboard.

There are many ways for a program to gain input and output, including

- File i/o, that is, reading and writing to files
- Console i/o, reading and writing to a console window, such as a terminal in UNIX-based operating systems or a DOS prompt in Windows.
- Network i/o, reading and writing from a network device
- String i/o, reading and writing treating a string as if it were the input or output device

While these may seem unrelated, they work very similarly. In fact, operating systems that follow the POSIX specification deal with files, devices, network sockets, consoles, and many other things all with one type of handle, a file descriptor. However, low-level interfaces provided by the operating system tend to be difficult to

48 HTTP://WWW.RESEARCH.ATT.COM/~{}BS/BS_FAQ2.HTML#OVERLOAD-DOT
49 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
50 Chapter 3.1.2 on page 45
51 Chapter 3.7.11 on page 274

use, so C++, like other languages, provide an abstraction to make programming easier. This abstraction is the **stream**.

4.8.1 Character encoding

American Standard Code for Information Interchange (ASCII) 95 chart

ASCII[52] is a CHARACTER-ENCODING SCHEME[53] based on the ORDERING[54] of the ENGLISH ALPHABET[55]. The 95 ASCII graphic characters numbered from 0x20 to 0x7E (32 to 126 decimal), also known as the printable characters, represent letters, digits, PUNCTUATION MARKS[56], and a few miscellaneous symbols. The first 32 ASCII characters, from 0x00 to 0x20, are known as control characters. The SPACE CHARACTER[57], that denotes the space between words, as produced by the space-bar of a keyboard, represented by code 0x20 (HEXADECIMAL[58]), is considered a non-printing graphic (or an invisible graphic) rather than a control character.

Binary	OCT[59]	DEC[60]	HEX[61]	GLYPH[62]
010 0000	040	32	20	SPACE[63]
010 0001	041	33	21	!64
010 0010	042	34	22	"65
010 0011	043	35	23	#66
010 0100	044	36	24	$67
010 0101	045	37	25	%68
010 0110	046	38	26	&69
010 0111	047	39	27	'70

52 HTTP://EN.WIKIPEDIA.ORG/WIKI/ASCII
53 HTTP://EN.WIKIPEDIA.ORG/WIKI/CHARACTER%20ENCODING
54 HTTP://EN.WIKIPEDIA.ORG/WIKI/ORDER%20%28MATHEMATICS%29
55 HTTP://EN.WIKIPEDIA.ORG/WIKI/ENGLISH%20ALPHABET
56 HTTP://EN.WIKIPEDIA.ORG/WIKI/PUNCTUATION%20MARKS
57 HTTP://EN.WIKIPEDIA.ORG/WIKI/SPACE%20%28PUNCTUATION%29
58 HTTP://EN.WIKIPEDIA.ORG/WIKI/HEXADECIMAL
63 HTTP://EN.WIKIPEDIA.ORG/WIKI/SPACE%20%28PUNCTUATION%29
64 HTTP://EN.WIKIPEDIA.ORG/WIKI/EXCLAMATION%20MARK
65 HTTP://EN.WIKIPEDIA.ORG/WIKI/QUOTATION%20MARK
66 HTTP://EN.WIKIPEDIA.ORG/WIKI/NUMBER%20SIGN
67 HTTP://EN.WIKIPEDIA.ORG/WIKI/DOLLAR%20SIGN
68 HTTP://EN.WIKIPEDIA.ORG/WIKI/PERCENT%20SIGN
69 HTTP://EN.WIKIPEDIA.ORG/WIKI/AMPERSAND
70 HTTP://EN.WIKIPEDIA.ORG/WIKI/APOSTROPHE

Binary	OCT[59]	DEC[60]	HEX[61]	GLYPH[62]
010 1000	050	40	28	(71
010 1001	051	41	29	)72
010 1010	052	42	2A	*73
010 1011	053	43	2B	+74
010 1100	054	44	2C	,75
010 1101	055	45	2D	-76
010 1110	056	46	2E	.77
010 1111	057	47	2F	/78
011 0000	060	48	30	079
011 0001	061	49	31	180
011 0010	062	50	32	281
011 0011	063	51	33	382
011 0100	064	52	34	483
011 0101	065	53	35	584
011 0110	066	54	36	685
011 0111	067	55	37	786
011 1000	070	56	38	887
011 1001	071	57	39	988
011 1010	072	58	3A	:89
011 1011	073	59	3B	;90
011 1100	074	60	3C	<91

71 http://en.wikipedia.org/wiki/Bracket
72 http://en.wikipedia.org/wiki/Bracket
73 http://en.wikipedia.org/wiki/Asterisk
74 http://en.wikipedia.org/wiki/Plus%20sign
75 http://en.wikipedia.org/wiki/Comma%20%28punctuation%29
76 http://en.wikipedia.org/wiki/Hyphen-minus
77 http://en.wikipedia.org/wiki/Full%20stop
78 http://en.wikipedia.org/wiki/Slash%20%28punctuation%29
79 http://en.wikipedia.org/wiki/0
80 http://en.wikipedia.org/wiki/1%20%28number%29
81 http://en.wikipedia.org/wiki/2%20%28number%29
82 http://en.wikipedia.org/wiki/3%20%28number%29
83 http://en.wikipedia.org/wiki/4%20%28number%29
84 http://en.wikipedia.org/wiki/5%20%28number%29
85 http://en.wikipedia.org/wiki/6%20%28number%29
86 http://en.wikipedia.org/wiki/7%20%28number%29
87 http://en.wikipedia.org/wiki/8%20%28number%29
88 http://en.wikipedia.org/wiki/9%20%28number%29
89 http://en.wikipedia.org/wiki/Colon%20%28punctuation%29
90 http://en.wikipedia.org/wiki/Semicolon
91 http://en.wikipedia.org/wiki/Less-than%20sign

Binary	Oct[59]	Dec[60]	Hex[61]	Glyph[62]
011 1101	075	61	3D	=[92]
011 1110	076	62	3E	>[93]
011 1111	077	63	3F	?[94]

Binary	Oct[95]	Dec[96]	Hex[97]	Glyph[98]
100 0000	100	64	40	@[99]
100 0001	101	65	41	A[100]
100 0010	102	66	42	B[101]
100 0011	103	67	43	C[102]
100 0100	104	68	44	D[103]
100 0101	105	69	45	E[104]
100 0110	106	70	46	F[105]
100 0111	107	71	47	G[106]
100 1000	110	72	48	H[107]
100 1001	111	73	49	I[108]
100 1010	112	74	4A	J[109]
100 1011	113	75	4B	K[110]
100 1100	114	76	4C	L[111]
100 1101	115	77	4D	M[112]
100 1110	116	78	4E	N[113]
100 1111	117	79	4F	O[114]
101 0000	120	80	50	P[115]

92 http://en.wikipedia.org/wiki/Equals%20sign
93 http://en.wikipedia.org/wiki/Greater-than%20sign
94 http://en.wikipedia.org/wiki/Question%20mark
99 http://en.wikipedia.org/wiki/%40
100 http://en.wikipedia.org/wiki/A
101 http://en.wikipedia.org/wiki/B
102 http://en.wikipedia.org/wiki/C
103 http://en.wikipedia.org/wiki/D
104 http://en.wikipedia.org/wiki/E
105 http://en.wikipedia.org/wiki/F
106 http://en.wikipedia.org/wiki/G
107 http://en.wikipedia.org/wiki/H
108 http://en.wikipedia.org/wiki/I
109 http://en.wikipedia.org/wiki/J
110 http://en.wikipedia.org/wiki/K
111 http://en.wikipedia.org/wiki/L
112 http://en.wikipedia.org/wiki/M
113 http://en.wikipedia.org/wiki/N
114 http://en.wikipedia.org/wiki/O
115 http://en.wikipedia.org/wiki/P

Binary	OCT[95]	DEC[96]	HEX[97]	GLYPH[98]
101 0001	121	81	51	Q[116]
101 0010	122	82	52	R[117]
101 0011	123	83	53	S[118]
101 0100	124	84	54	T[119]
101 0101	125	85	55	U[120]
101 0110	126	86	56	V[121]
101 0111	127	87	57	W[122]
101 1000	130	88	58	X[123]
101 1001	131	89	59	Y[124]
101 1010	132	90	5A	Z[125]
101 1011	133	91	5B	[[126]
101 1100	134	92	5C	\[127]
101 1101	135	93	5D	][128]
101 1110	136	94	5E	^[129]
101 1111	137	95	5F	_[130]

Binary	OCT[131]	DEC[132]	HEX[133]	GLYPH[134]
110 0000	140	96	60	`[135]
110 0001	141	97	61	A[136]
110 0010	142	98	62	B[137]
110 0011	143	99	63	C[138]
110 0100	144	100	64	D[139]

116 HTTP://EN.WIKIPEDIA.ORG/WIKI/Q
117 HTTP://EN.WIKIPEDIA.ORG/WIKI/R
118 HTTP://EN.WIKIPEDIA.ORG/WIKI/S
119 HTTP://EN.WIKIPEDIA.ORG/WIKI/T
120 HTTP://EN.WIKIPEDIA.ORG/WIKI/U
121 HTTP://EN.WIKIPEDIA.ORG/WIKI/V
122 HTTP://EN.WIKIPEDIA.ORG/WIKI/W
123 HTTP://EN.WIKIPEDIA.ORG/WIKI/X
124 HTTP://EN.WIKIPEDIA.ORG/WIKI/Y
125 HTTP://EN.WIKIPEDIA.ORG/WIKI/Z
126 HTTP://EN.WIKIPEDIA.ORG/WIKI/BRACKET
127 HTTP://EN.WIKIPEDIA.ORG/WIKI/BACKSLASH
128 HTTP://EN.WIKIPEDIA.ORG/WIKI/BRACKET
129 HTTP://EN.WIKIPEDIA.ORG/WIKI/CARET
130 HTTP://EN.WIKIPEDIA.ORG/WIKI/UNDERSCORE
135 HTTP://EN.WIKIPEDIA.ORG/WIKI/GRAVE%20ACCENT
136 HTTP://EN.WIKIPEDIA.ORG/WIKI/A
137 HTTP://EN.WIKIPEDIA.ORG/WIKI/B
138 HTTP://EN.WIKIPEDIA.ORG/WIKI/C
139 HTTP://EN.WIKIPEDIA.ORG/WIKI/D

Binary	OCT[131]	DEC[132]	HEX[133]	GLYPH[134]
110 0101	145	101	65	E[140]
110 0110	146	102	66	F[141]
110 0111	147	103	67	G[142]
110 1000	150	104	68	H[143]
110 1001	151	105	69	I[144]
110 1010	152	106	6A	J[145]
110 1011	153	107	6B	K[146]
110 1100	154	108	6C	L[147]
110 1101	155	109	6D	M[148]
110 1110	156	110	6E	N[149]
110 1111	157	111	6F	O[150]
111 0000	160	112	70	P[151]
111 0001	161	113	71	Q[152]
111 0010	162	114	72	R[153]
111 0011	163	115	73	S[154]
111 0100	164	116	74	T[155]
111 0101	165	117	75	U[156]
111 0110	166	118	76	V[157]
111 0111	167	119	77	W[158]
111 1000	170	120	78	X[159]
111 1001	171	121	79	Y[160]

140 HTTP://EN.WIKIPEDIA.ORG/WIKI/E
141 HTTP://EN.WIKIPEDIA.ORG/WIKI/F
142 HTTP://EN.WIKIPEDIA.ORG/WIKI/G
143 HTTP://EN.WIKIPEDIA.ORG/WIKI/H
144 HTTP://EN.WIKIPEDIA.ORG/WIKI/I
145 HTTP://EN.WIKIPEDIA.ORG/WIKI/J
146 HTTP://EN.WIKIPEDIA.ORG/WIKI/K
147 HTTP://EN.WIKIPEDIA.ORG/WIKI/L
148 HTTP://EN.WIKIPEDIA.ORG/WIKI/M
149 HTTP://EN.WIKIPEDIA.ORG/WIKI/N
150 HTTP://EN.WIKIPEDIA.ORG/WIKI/O
151 HTTP://EN.WIKIPEDIA.ORG/WIKI/P
152 HTTP://EN.WIKIPEDIA.ORG/WIKI/Q
153 HTTP://EN.WIKIPEDIA.ORG/WIKI/R
154 HTTP://EN.WIKIPEDIA.ORG/WIKI/S
155 HTTP://EN.WIKIPEDIA.ORG/WIKI/T
156 HTTP://EN.WIKIPEDIA.ORG/WIKI/U
157 HTTP://EN.WIKIPEDIA.ORG/WIKI/V
158 HTTP://EN.WIKIPEDIA.ORG/WIKI/W
159 HTTP://EN.WIKIPEDIA.ORG/WIKI/X
160 HTTP://EN.WIKIPEDIA.ORG/WIKI/Y

Binary	OCT[131]	DEC[132]	HEX[133]	GLYPH[134]
111 1010	172	122	7A	z[161]
111 1011	173	123	7B	{[162]
111 1100	174	124	7C	|[163]
111 1101	175	125	7D	}[164]
111 1110	176	126	7E	~[165]

166

4.8.2 Streams

A stream is a type of object from which we can take values, or to which we can pass values. This is done transparently in terms of the underlying code that demonstrates the use of the `std::cout` stream, known as the *standard output stream*.

```
// 'Hello World!' program

#include <iostream>

int main()
{
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

Almost all input and output one ever does can be modeled very effectively as a stream. Having one common model means that one only has to learn it once. If you understand streams, you know the basics of how to output to files, the screen, sockets, pipes, and anything else that may come up.

A stream is an object that allows one to push data in or out of a medium, in order. Usually a stream can only output or can only input. It is possible to have a stream that does both, but this is rare. One can think of a stream as a car driving along a one-way street of information. An output stream can insert data and move on. It (usually) cannot go back and adjust something it has already written. Similarly, an input stream can read the next bit of data and then wait for the one that comes after it. It does not skip data or rewind and see what it had read 5 minutes ago.

161 HTTP://EN.WIKIPEDIA.ORG/WIKI/Z
162 HTTP://EN.WIKIPEDIA.ORG/WIKI/BRACKET
163 HTTP://EN.WIKIPEDIA.ORG/WIKI/VERTICAL%20BAR
164 HTTP://EN.WIKIPEDIA.ORG/WIKI/BRACKET
165 HTTP://EN.WIKIPEDIA.ORG/WIKI/TILDE
166 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

The semantics of what a stream's read and write operations do depend on the type of stream. In the case of a file, an input file stream reads the file's contents in order without rewinding, and an output file stream writes to the file in order. For a console stream, output means displaying text, and input means getting input from the user via the console. If the user has not inputted anything, then the program *blocks*, or waits, for the user to enter in something.

iostream

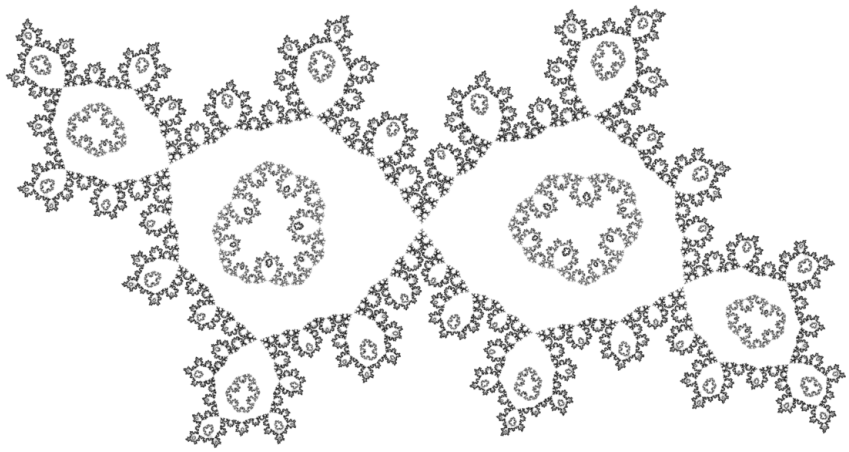


Figure 24: c++ program that uses iostream to save output to the file

`iostream` is a HEADER FILE[167] used for input/output. Part of the C++ standard library. The name stands for **I**nput/**O**utput **Stream**. In C++ there is no special syntax for streaming data input or output. Instead, these are combined as a LIBRARY[168] of functions. Like we have seen with the C STANDARD LIBRARY USE OF `<cstdio>` HEADER[169], `iostream` provides basic OOP services for I/O.

The <iostream> automatically defines and uses a few standard objects:

- `cin`, an object of the istream class that reads data from the standard input device.
- `cout`, an object of the ostream class, which displays data to the standard output device.

167 Chapter 3.1.6 on page 53
168 Chapter 6.3.3 on page 584
169 Chapter 3.7.11 on page 274

- `cerr`, another object of the ostream class that writes unbuffered output to the standard error device.
- `clog`, like cerr, but uses buffered output.

for sending data to and from the STANDARD STREAMS[170] input, output, error (unbuffered), and error (buffered) respectively. As part of the C++ standard library, these objects are a part of the *std namespace*.

Standard input, output, and error

The most common streams one uses are `cout`, `cin`, and `cerr` (pronounced "c out", "c in", and "c err(or)", respectively). They are defined in the header `<iostream>`. Usually, these streams read and write from a console or terminal. In UNIX-based operating systems, such as Linux and Mac OS X, the user can redirect them to other files, or even other programs, for logging or other purposes. They are analogous to `stdout`, `stdin`, and `stderr` found in C. `cout` is used for generic output, `cin` is used for input, and `cerr` is used for printing errors. (`cerr` typically goes to the same place as `cout`, unless one or both is redirected, but it is not buffered and allows the user to fine-tune which parts of the program's output is redirected where.)

Output

The standard syntax for outputting to a stream, in this case, `cout`, is

```
cout << some_data << some_more_data;
```

Example

```
#include <iostream>

using namespace std;

int main()
{
  int a = 1;
  cout << "Hello world! " << a << '\n';

  return 0;
}
```

Result of Execution

170 HTTP://EN.WIKIPEDIA.ORG/WIKI/STANDARD%20STREAMS

```
Hello world! 1
```

To add a line break, send a newline character, `\n` or use `std::endl`, which writes a newline and flushes the stream's buffer.

Example

```
#include <iostream>
#include <ostream>

using namespace std;

int main()
{
  int a = 1;
  char x = 13;
  cout << "Hello world!" << "\n" << a << endl << x << endl;

  return 0;
}
```

Execution

```
Hello world!
1
```

It is always a good idea to end your output with a blank line, so as to not mess up with user's terminals.

As seen in the "Hello World!" program, we direct the output to `std::cout`. This means that it is a *member* of the *standard library*. For now, don't worry about what this means; we will cover the library and namespaces in later chapters.

What you do need to remember is that, in order to use the output stream, you must include a reference to the standard IO library, as shown here: `#include <iostream>`

This opens up a number of streams, functions and other programming devices which we can now use. For this section, we are interested in two of these; `std::cout` and `std::endl`.

Once we have referenced the standard IO library, we can use the output stream very simply. To use a stream, give its name, then *pipe* something in or out of it, as shown: `std::cout << "Hello, world!";`

The << operator feeds everything to the right of it into the stream. We have essentially fed a text object into the stream. That's as far as our work goes; the stream

now decides what to do with that object. In the case of the output stream, it's printed on-screen.

We're not limited to only sending a single object type to the stream, nor indeed are we limited to one object a time. Consider the examples below:

```
std::cout << "Hello, " << "Joe"<< std::endl;
std::cout << "The answer to life, the universe and everything is " << 42 <<
std::endl;
```

As can be seen, we feed in various values, separated by a pipe character. The result comes out something like:

```
Hello, Joe
The answer to life, the universe and everything is 42
```

You will have noticed the use of `std::endl` throughout some of the examples so far. This is the newline constant. It is a member of the standard IO library, and comes "free" when we instantiate that in order to use the output stream. When the output stream receives this constant, it starts a new line in the console.

And of course, we're not limited to sending only ONE newline, either:

```
std::cout << "Hello, " << "Joe" << std::endl << std::endl;
std::cout << "How old are you?";
```

Which produces something like:

```
Hello, Joe

How old are you?
```

Input

What would be the use of an application that only ever outputted information, but didn't care about what its users wanted? Minimal to none. Fortunately, inputting is as easy as outputting when you're using the stream.

The *standard input stream* is called `std::cin` and is used very similarly to the output stream. Once again, we instantiate the standard IO library:

```
#include <iostream>
```

This gives us access to `std::cin` (and the rest of that class). Now, we give the name of the stream as usual, and pipe output from it into a variable. A number of things have to happen here, demonstrated in the example below:

```
#include <iostream>
int main(int argc, char argv[]) {
  int a;
  std::cout << "Hello! How old are you? ";
  std::cin >> a;
  std::cout << "You're really " << a << " years old?" << std::endl;

  return 0;
}
```

We instantiate the standard IO library as usual, and call our main function in the normal way. Now we need to consider where the user's input goes. This calls for a variable (discussed in a later chapter) which we declare as being called `a`.

Next, we send some output, asking the user for their age. The real input happens now; everything the user types until they hit Enter is going to be stored in the input stream. We pull this out of the input stream and save it in our variable.

Finally, we output the user's age, piping the contents of our variable into the output stream.

Note: You will notice that if anything other than a whole number is entered, the program will crash. This is due to the way in which we set up our variable. Don't worry about this for now; we will cover variables later on.

A Program Using User Input

The following program inputs two numbers from the user and prints their sum:

```
#include <iostream>

int main()
{
   int num1, num2;
   std::cout << "Enter number 1: ";
   std::cin >> num1;
   std::cout << "Enter number 2: ";
   std::cin >> num2;
   std::cout << "The sum of " << num1 << " and " << num2 << " is "
             << num1 + num2 << ".\n";
   return 0;
}
```

Just like `std::cout` which represents the standard output stream, the C++ library provides (and the `iostream` header declares) the object `std::cin` representing standard input, which usually gets input from the keyboard. The statement:

```
std::cin >> num1;
```

uses the *extraction operator* (>>) to get an integer input from the user. When used to input integers, any leading whitespace is skipped, a sequence of valid digits optionally preceded by a + or − sign is read and the value stored in the variable. Any remaining characters in the user input are not *consumed*. These would be considered next time some input operation is performed.

If you want the program to use a function from a specific namespace, normally you must specify which namespace the function is in. The above example calls to `cout`, which is a member of the `std` namespace (hence `std::cout`). If you want a program to specifically use the std namespace for an identifier, which essentially removes the need for all future scope resolution (e.g. `std::`), you could write the above program like this:

```
#include <iostream>

using namespace std;

int main()
{
    int num1, num2;
    cout << "Enter number 1: ";
    cin >> num1;
    cout << "Enter number 2: ";
    cin >> num2;
    cout << "The sum of " << num1 << " and " << num2 << " is "
              << num1 + num2 << ".\n";
    return 0;
}
```

Please note that 'std' namespace is the namespace defined by standard C++ library.

Manipulators

A manipulator is a function that can be passed as an argument to a stream in different circumstances. For example, the manipulator 'hex' will cause the stream object to format subsequent integer input to the stream in hexadecimal instead of decimal. Likewise, 'oct' results in integers displaying in octal, and 'dec' reverts back to decimal.

Example

```
#include <iostream>
using namespace std;

int main()
{
```

```
  cout << dec << 16 << ' ' << 10 << endl;
  cout << oct << 16 << ' ' << 10 << endl;
  cout << hex << 16 << ' ' << 10 << endl;

  return 0;
}
```

Execution

```
  16 10
  20 12
  10 a
```

There are many manipulators which can be used in conjunction with streams to simplify the formatting of input. For example, 'setw()' sets the field width of the data item next displayed. Used in conjunction with 'left' and 'right'(which set the justification of the data), 'setw' can easily be used to create columns of data.

Example

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
        cout << setw(10) << right << 90 << setw(8) << "Help!" << endl;
        cout << setw(10) << left << 45 << setw(8) << "Hi!" << endl;

        return 0;
}
```

Execution

```
         90   Help!
  45       Hi!
```

The data in the top row display at the right of the columns created by 'setw', while in the next row, the data is left justified in the column. Please note the inclusion of a new library 'iomanip'. Most formatting manipulators require this library.

Here are some other manipulators and their uses:

Manipulator	**Function**
boolalpha	displays boolean values as 'true' and 'false' instead of as integers.
noboolalpha	forces bools to display as integer values

Manipulator	Function
showuppercase	converts strings to uppercase before displaying them
noshowuppercase	displays strings as they are received, instead of in uppercase
fixed	forces floating point numbers to display with a fixed number of decimal places
scientific	displays floating point numbers in scientific notation

Buffers

Most stream objects, including 'cout' and 'cin', have an area in memory where the information they are transferring sits until it is asked for. This is called a 'buffer'. Understanding the function of buffers is essential to mastering streams and their use.

Example

```
#include <iostream>
using namespace std;

int main()
{
  int num1, num2;
  cin >> num1;
  cin >> num2;

  cout << "Number1: " << num1 << endl
       << "Number2: " << num2 << endl;

  return 0;
}
```

Execution 1

```
>74
>27
Number1: 74
Number2: 27
```

The inputs are given separately, with a hard return between them. '>' denotes user input.

Execution 2

```
  >74 27
  Number1: 74
  Number2: 27
```

The inputs are entered on the same line. They both go into the 'cin' stream buffer, where they are stored until needed. As 'cin' statements are executed, the contents of the buffer are read into the appropriate variables.

Execution 3

```
  >74 27 56
  Number1: 74
  Number2: 27
```

In this example, 'cin' received more input than it asked for. The third number it read in, 56, was never inserted into a variable. It would have stayed in the buffer until 'cin' was called again. The use of buffers can explain many strange behaviors that streams can exhibit.

Example

```
#include <iostream>
using namespace std;

int main()
{
  int num1, num2, num3;
  cin >> num1 >> num2;

  cout << "Number1: " << num1 << endl
    << "Number2: " << num2 << endl;

  cin >> num3;

  cout << "Number3: " << num3 << endl;

  return 0;
}
```

Execution

```
  >45 89 37
  Number1: 45
  Number2: 89
  Number3: 37
```

Notice how all three numbers were entered at the same time in one line, but the stream only pulled them out of the buffer when they were asked for. This can cause unexpected output, since the user might accidentally put an extra space into his input. A well written program will test for this type of unexpected input and handle it gracefully.

ios

ios is a HEADER FILE[171] in the C++ standard library which defines several types and functions basic to the operation of iostreams. This header is typically included automatically by other iostream headers. Programmers rarely include it directly.

Typedefs

Name	description
`ios`	Supports the `ios` class from the old `iostream` library.
`streamoff`	Supports internal operations.
`streampos`	Holds the current position of the buffer pointer or file pointer.
`streamsize`	Specifies the size of the stream.
`wios`	Supports the `wios` class from the old `iostream` library.
`wstreampos`	Holds the current position of the buffer pointer or file pointer.

Manipulators

Name	description
`boolalpha`	Specifies that variables of type `bool` appear as true or false in the stream.
`dec`	Specifies that integer variables appear in base 10 notation.

171 Chapter 3.1.6 on page 53

Name	description
`fixed`	Specifies that a floating-point number is displayed in fixed-decimal notation.
`hex`	Specifies that integer variables appear in base 16 notation.
`internal`	Causes a number's sign to be left justified and the number to be right justified.
`left`	Causes text that is not as wide as the output width to appear in the stream flush with the left margin.
`noboolalpha`	Specifies that variables of type `bool` appear as 1 or 0 in the stream.
`noshowbase`	Turns off indicating the notational base in which a number is displayed.
`noshowpoint`	Displays only the whole-number part of floating-point numbers whose fractional part is zero.
`noshowpos`	Causes positive numbers to not be explicitly signed.
`noskipws`	Cause spaces to be read by the input stream.
`nounitbuf`	Causes output to be buffered and processed when the buffer is full.
`nouppercase`	Specifies that hexadecimal digits and the exponent in scientific notation appear in lowercase.
`oct`	Specifies that integer variables appear in base 8 notation.
`right`	Causes text that is not as wide as the output width to appear in the stream flush with the right margin.
`scientific`	Causes floating point numbers to be displayed using scientific notation.
`showbase`	Indicates the notational base in which a number is displayed.
`showpoint`	Displays the whole-number part of a floating-point number and digits to the right of the decimal point even when the fractional part is zero.

Name	description
`showpos`	Causes positive numbers to be explicitly signed.
`skipws`	Cause spaces to not be read by the input stream.
`unitbuf`	Causes output to be processed when the buffer is not empty.
`uppercase`	Specifies that hexadecimal digits and the exponent in scientific notation appear in uppercase.

Classes

Name	description
`basic_ios`	The template class describes the storage and member functions common to both input streams (of template class basic_istream) and output streams (of template class basic_ostream) that depend on the template parameters.
`fpos`	The template class describes an object that can store all the information needed to restore an arbitrary file-position indicator within any stream.
`ios_base`	The class describes the storage and member functions common to both input and output streams that do not depend on the template parameters.

fstream

With `cout` and `cin`, we can do basic communication with the user. For more complex io, we would like to read from and write to files. This is done with file stream classes, defined in the header **<fstream>**. `ofstream` is an output file stream, and `ifstream` is an input file stream.

Files

To *open* a file, one can either call `open` on the file stream or, more commonly, use the constructor. One can also supply an open mode to further control the file stream. Open modes include

- ios::app Leaves the file's original contents and appends new data to the end.
- ios::out Outputs new data in the file, removing the old contents. (default for ofstream)
- ios::in Reads data from the file. (default for ifstream)

Example

```
// open a file called Test.txt and write "HELLO, HOW ARE YOU?" to it
#include <fstream>

using namespace std;

int main()
{
  ofstream file1;

  file1.open("file1.txt", ios::app);
  file1 << "This data will be appended to the file file1.txt\n";
  file1.close();

  ofstream file2("file2.txt");
  file2 << "This data will replace the contents of file2.txt\n";

  return 0;
}
```

The call to close() can be omitted if you do not care about the return value (whether it succeeded); the destructors will call close when the object goes out of scope.

If an operation (e.g. opening a file) was unsuccessful, a flag is set in the stream object. You can check the flags' status using the bad() or fail() member functions, which return a boolean value. The stream object doesn't throw any exceptions in such a situation; hence manual status check is required. See reference for details on bad() and fail().

Text input until EOF/error/invalid input

Input from the stream `infile` to a variable `data` until one of the following:

- EOF reached on `infile`.
- An error occurs while reading from `infile` (e.g., connection closed while reading from a remote file).
- The input item is invalid, e.g. non-numeric characters, when `data` is of type **int**.

```
#include <iostream>
```

```
// ...

while (infile >> data)
{
    // manipulate data here
}
```

Note that the following is not correct:

```
#include <iostream>

// ...

while (!infile.eof())
{
    infile >> data; // wrong!
    // manipulate data here
}
```

This will cause the last item in the input file to be processed twice, because `eof()` does not return true until input *fails* due to EOF.

ostream

Classes and output streams

It is often useful to have your own classes' instances compatible with the stream framework. For instance, if you defined the class Foo like this:

```
class Foo
{
public:

       Foo() : x(1), y(2)
       {
       }

       int x, y;
};
```

You will not be able to pass its instance to cout directly using the '<<' operator, because it is not defined for these two objects (Foo and ostream). What needs to be done is to define this operator and thus bind the user-defined class with the stream class.

```
ostream& operator<<(ostream& output, Foo& arg)
{
       output << arg.x << "," << arg.y;
```

```
        return output;
}
```

Now this is possible:

```
Foo my_object;
cout << "my_object's values are: " << my_object << endl;
```

The operator function needs to have 'ostream&' as its return type, so chaining output works as usual between the stream and objects of type Foo:

```
Foo my1, my2, my3;
cout << my1 << my2 << my3;
```

This is because (cout << my1) is of type ostream&, so the next argument (my2) can be appended to it in the same expression, which again gives an ostream& so my3 can be appended and so on.

If you decided to restrict access to the member variables x and y (which is probably a good idea) within the class Foo, i.e.:

```
class Foo
{
public:

        Foo() : x(1), y(2)
        {
        }

private:
        int x, y;
};
```

you will have trouble, because the global operator<< function doesn't have access to the private variables of its second argument. There are two possible solutions to this problem:

1. Within the class Foo, declare the operator<< function as the classes' friend which grants it access to private members, i.e. add the following line to the class declaration:

```
friend ostream& operator<<(ostream& output, Foo& arg);
```

Then define the operator<< function as you normally would (note that the declared function is not a member of Foo, just its friend, so don't try defining it as Foo::operator<<).

2. Add public-available functions for accessing the member variables and make the operator<< function use these instead:

```
class Foo
{
public:

	Foo() : x(1), y(2)
	{
	}

	int get_x()
	{
		return x;
	}

	int get_y()
	{
		return y;
	}

private:
	int x, y;
};

ostream& operator<<(ostream& output, Foo& arg)
{
	output << arg.get_x() << "," << arg.get_y();
	return output;
}
```

I[172]

4.8.3 The string class

The string class is a part of the C++ standard library, used for convenient manipulation of sequences of characters, to replace the static, unsafe C method of handling strings. To use the string class in a program, the <string> header must be included. The standard library string class can be accessed through the *std namespace*.

The basic template class is `basic_string<>` and its standard specializations are `string` and `wstring`.

Basic usage

Declaring a *std* string is done by using one of these two methods:

```
using namespace std;
```

172 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A

```
string std_string;
```

or

```
std::string std_string;
```

Text I/O

This section will deal only with keyboard and text input. There are many other inputs that can be read (mouse movements and button clicks, etc...), but these will not be covered in this section, even reading the special keys of the keyboard will be excluded.

Perhaps the most basic use of the string class is for reading text from the user and writing it to the screen. In the header file `iostream`, C++ defines an object named cin that handles input in much the same way that cout handles output.

```
// snipped designed to get an integer value from the user
int x;
std::cin >> x;
```

The >> operator will cause the execution to stop and will wait for the user to type something. If the user types a valid integer, it will be converted into an integer value and stored in x.

If the user types something other than an integer, the compiler will not report an error. Instead, it leaves the old content (a "random" meaningless value) in `x` and continues.

This can then be extended into the following program:

```
#include <iostream>
#include <string>

int main(){
    std::string name;
    std::cout << "Please enter your first name: ";
    std::cin >> name;
    std::cout << "Welcome " << name << "!" << std::endl;

    return 0;
}
```

Although a string may hold a sequence containing any character--including spaces and nulls--when reading into a string using cin and the extraction operator (>>) only the characters before the first space will be stored. Alternatively, if an entire line of text is desired, the *getline* function may be used:

```
    std::getline(std::cin, name);
```

Getting user input

Fortunately, there is a way to check and see if an input statement succeeds. We can invoke the good function on cin to check what is called the stream state. good returns a bool: if true, then the last input statement succeeded. If not, we know that some previous operation failed, and also that the next operation will fail.

Thus, getting input from the user might look like this:

```
#include <iostream>
int main ()
{
  using namespace std; // pull in the std namespace
  int x;

  // prompt the user for input
  cout << "Enter an integer: ";

  // get input
  cin >> x;

  // check and see if the input statement succeeded
  if (cin.good() == false) {
    cout << "That was not an integer." << endl;
    return -1;
  }

  // print the value we got from the user
  cout << x << endl;
  return 0;
}
```

cin can also be used to input a string:

```
string name;

cout << "What is your name? ";
cin >> name;
cout << name << endl;
```

As with the scanf() function from the Standard C Library, this statement only takes the first word of input, and leaves the rest for the next input statement. So, if you run this program and type your full name, it will only output your first name.

You may also notice the >> operator doesn't handle errors as expected (for example, if you accidentally typed your name in a prompt for a number.) Because of these issues, it may be more suitable to read a line of text, and using the line for input — this is performed using the function called getline.

```
string name;

cout << "What is your name? ";
getline (cin, name);
cout << name << endl;
```

The first argument to getline is cin, which is where the input is coming from. The second argument is the name of the string variable where you want the result to be stored.

getline reads the entire line until the user hits Return or Enter. This is useful for inputting strings that contain spaces.

In fact, getline is generally useful for getting input of any kind. For example, if you wanted the user to type an integer, you could input a string and then check to see if it is a valid integer. If so, you can convert it to an integer value. If not, you can print an error message and ask the user to try again.

To convert a string to an integer you can use the strtol function defined in the header file cstdlib. (Note that the older function atoi is less safe than strtol, as well as being less capable.)

If you still need the features of the >> operator, you will need to create a string stream as available from `<sstream>`. The use of this stream will be discussed in a later chapter.

More advanced string manipulation

We will be using this dummy string for some of our examples.

```
string str("Hello World!");
```

This invokes the default constructor with a `const char*` argument. Default constructor creates a string which contains nothing, i.e. no characters, not even a `'\0'` (however std::string is not null terminated).

```
string str2(str);
```

Will trigger the copy constructor. `std::string` knows enough to make a deep copy of the characters it stores.

```
string str2 = str;
```

This will copy strings using assignment operator. Effect of this code is same as using copy constructor in example above.

Size

```
string::size_type string::size() const;
string::size_type string::length() const;
```

So for example one might do:

```
string::size_type strSize =  str.size();
string::size_type strSize2 = str2.length();
```

The methods `size()` and `length()` both return the size of the string object. There is no apparent difference. Remember that the last character in the string is `size() - 1` and not `size()`. Like in C-style strings, and arrays in general, `std::string` starts counting from 0.

I/O

```
ostream& operator<<(ostream &out, string &str);
istream& operator>>(istream &in, string &str);
```

The shift operators (>> and <<) have been overloaded so you can perform I/O operations on `istream` and `ostream` objects, most notably `cout`, `cin`, and filestreams. Thus you could just do console I/O like this:

```
std::cout << str << endl;
std::cin >> str;

istream& getline (istream& in, string& str, char delim = '\n');
```

Alternatively, if you want to read entire lines at a time, use `getline()`. Note that this is not a member function. `getline()` will retrieve characters from input stream `in` and assign them to `str` until `EOF` is reached or `delim` is encountered. `getline` will reset the input string before appending data to it. `delim` can be set to any `char` value and acts as a general delimiter. Here is some example usage:

```
#include <fstream>
//open a file
std::ifstream file("somefile.cpp");
std::string data, temp;

while( getline(file, temp, '#')) //while data left in file
{
    //append data
    data += temp;
}
```

```
std::cout << data;
```

Because of the way `getline` works (i.e. it returns the input stream), you can nest multiple `getline()` calls to get multiple strings; however this may significantly reduce readability.

Operators

```
char& string::operator[](string::size_type pos);
```

`Chars` in `string`s can be accessed directly using the overloaded subscript (`[]`) operator, like in `char` arrays:

```
std::cout << str[0] << str[2];
```

prints "Hl".

`std::string` supports casting from the older C string type `const char*`. You can also assign or append a simple `char` to a string. Assigning a `char*` to a `string` is as simple as

```
str = "Hello World!";
```

If you want to do it character by character, you can also use

```
str = 'H';
```

Not surprisingly, `operator+` and `operator+=` are also defined! You can append another `string`, a `const char*` or a `char` to any string.

The comparison operators `>`, `<`, `==`, `>=`, `<=`, `!=` all perform comparison operations on strings, similar to the C strcmp() function. These return a true/false value.

```
if(str == "Hello World!")
{
 std::cout << "Strings are equal!";
}
```

Searching strings

```
string::size_type string::find(string needle, string::size_type pos = 0) const;
```

You can use the `find()` member function to find the first occurrence of a string inside another. `find()` will look for `needle` inside `this` starting from position `pos` and return the position of the first occurrence of the `needle`. For example:

```
std::string haystack = "Hello World!";
std::string needle = "o";
std::cout << haystack.find(needle);
```

Will simply print "4" which is the index of the first occurrence of "o" in `str`. If we want the "o" in "World", we need to modify `pos` to point past the first occurrence. `str.find(find, 4)` would return 4, while `str.find(find, 5)` would give 7. If the substring isn't found, `find()` returns `std::string::npos`.This simple code searches a string for all occurrences of "wiki" and prints their positions:

```
std::string wikistr = "wikipedia is full of wikis (wiki-wiki means fast)";
for(string::size_type i = 0, tfind; (tfind = wikistr.find("wiki", i)) !=
 string::npos; i = tfind + 1)
{
 std::cout << "Found occurrence of 'wiki' at position " << tfind << std::endl;
}

string::size_type string::rfind(string needle, string::size_type pos =
 string::npos) const;
```

The function `rfind()` works similarly, except it returns the *last* occurrence of the passed string.

Inserting/erasing

```
string& string::insert(size_type pos, const string& str);
```

You can use the `insert()` member function to insert another string into a string. For example:

```
string newstr = " Human";
str.insert (5,newstr);
```

Would return *Hello Human World!*

```
string& string::erase(size_type pos, size_type n);
```

You can use `erase()` to remove a substring from a string. For example:

```
str.erase (6,11);
```

Would return *Hello!*

```
string& string::substr(size_type pos, size_type n);
```

You can use `substr()` to extract a substring from a string. For example:

```
string str = "Hello World!";
string part = str.substr(6,5);
```

Would return *World*.

Backwards compatibility

```
const char* string::c_str() const;
const char* string::data() const;
```

For backwards compatibility with C/C++ functions which only accept `char*` parameters, you can use the member functions `string::c_str()` and `string::data()` to return a temporary `const char*` string you can pass to a function. The difference between these two functions is that `c_str()` returns a null-terminated string while `data()` does not necessarily return a null-terminated string. So, if your legacy function requires a null-terminated string, use `c_str()`, otherwise use `data()` (and presumably pass the length of the string in as well).

String Formatting

Strings can only be appended to other strings, but not to numbers or other datatypes, so something like `std::string("Foo") + 5` would not result in a string with the content `"Foo5"`. To convert other datatypes into string there exist the class `std::ostringstream`, found in the include file `<sstream>`. `std::ostringstream` acts exactly like `std::cout`, the only difference is that the output doesn't go to the current standard output as provided by the operating system, but into an internal buffer, that buffer can be converted into a `std::string` via the `std::ostringstream::str()` method.

Example

```
#include <iostream>
#include <sstream>

int main()
{
    std::ostringstream buffer;
```

```
    // Use the std::ostringstream just like std::cout or other iostreams
    buffer << "You have: " << 5 << " Helloworlds in your inbox";

    // Convert the std::ostringstream to a normal string
    std::string text = buffer.str();

    std::cout << text << std::endl;

    return 0;
}
```

Advanced use

173

4.9 Chapter Summary

1. STRUCTURES[174]
2. UNIONS[175]
3. CLASSES[176] (INHERITANCE[177], MEMBER FUNCTIONS[178], POLYMORPHISM[179] and THIS[180] pointer)
   a) ABSTRACT CLASSES[181] including PURE ABSTRACT CLASSES (ABSTRACT TYPES) [182]
   b) NICE CLASS[183]
4. OPERATOR OVERLOADING[184]
5. STANDARD INPUT/OUTPUT STREAMS LIBRARY[185]
   a) STRING[186]

173 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
174 Chapter 4 on page 385
175 Chapter 4.1.2 on page 390
176 Chapter 4.2.3 on page 393
177 Chapter 4.3.2 on page 398
178 Chapter 4.3.4 on page 407
179 Chapter 4.3.5 on page 418
180 Chapter 4.3.4 on page 405
181 Chapter 4.3.12 on page 430
182 Chapter 4.3.13 on page 432
183 Chapter 4.3.13 on page 434
184 Chapter 4.6 on page 438
185 Chapter 4.7.3 on page 451
186 Chapter 4.8.2 on page 473

3[187]

3[188]

187 http://en.wikibooks.org/wiki/Category%3A

188 http://en.wikibooks.org/wiki/Category%3AC%2B%2B%20Programming

5 Advanced Features

5.1 Templates

Templates are a way to make code more reusable. Trivial examples include creating generic data structures which can store arbitrary data types. Templates are of great utility to programmers, especially when combined with multiple INHERITANCE[1] and OPERATOR OVERLOADING[2]. The STANDARD TEMPLATE LIBRARY[3] (STL) provides many useful functions within a framework of connected templates.

As the templates are very expressive they may be used for things other than generic programming. One such use is called TEMPLATE METAPROGRAMMING[4], which is a way of pre-evaluating some of the code at compile-time rather than run-time. Further discussion here only relates to templates as a method of generic programming.

By now you should have noticed that functions that perform the same tasks tend to look similar. For example, if you wrote a function that prints an int, you would have to have the int declared first. This way, the possibility of error in your code is reduced, however, it gets somewhat annoying to have to create different versions of functions just to handle all the different data types you use. For example, you may want the function to simply print the input variable, regardless of what type that variable is. Writing a different function for every possible input type (`double`,`char *`, etc. ...) would be extremely cumbersome. That is where templates come in.

Templates solve some of the same problems as macros, generate "optimized" code at compile time, but are subject to C++'s strict type checking.

Parameterized types, better known as templates, allow the programmer to create one function that can handle many different types. Instead of having to take into

1 HTTP://EN.WIKIPEDIA.ORG/WIKI/INHERITANCE%20IN%20OBJECT-ORIENTED%20PROGRAMMING

2 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPERATOR%20OVERLOADING

3 Chapter 5.1.5 on page 499

4 HTTP://EN.WIKIPEDIA.ORG/WIKI/TEMPLATE%20METAPROGRAMMING

account every data type, you have one arbitrary parameter name that the compiler then replaces with the different data types that you wish the function to use, manipulate, etc.

- Templates are instantiated at compile-time with the source code.
- Templates are type safe.
- Templates allow user-defined specialization.
- Templates allow non-type parameters.
- Templates use “lazy structural constraints”.
- Templates support mix-ins.

Syntax for Templates

Templates are pretty easy to use, just look at the syntax:

```
template <class TYPEPARAMETER>
```

(or, equivalently, and preferred by some)

```
template <typename TYPEPARAMETER>
```

5.1.1 Function template

There are two kinds of templates. A *function template* behaves like a function that can accept arguments of many different types. For example, the Standard Template Library contains the function template `max(x, y)` which returns either `x` or `y`, whichever is larger. `max()` could be defined like this:

```
template <typename TYPEPARAMETER>
TYPEPARAMETER max(TYPEPARAMETER x, TYPEPARAMETER y)
{
    if (x < y)
        return y;
    else
        return x;
}
```

This template can be called just like a function:

```
std::cout << max(3, 7);   // outputs 7
```

The compiler determines by examining the arguments that this is a call to `max(int, int)` and *instantiates* a version of the function where the type `TYPEPARAMETER` is `int`.

This works whether the arguments `x` and `y` are integers, strings, or any other type for which it makes sense to say `x < y`". If you have defined your own data type, you can use operator overloading to define the meaning of < for your type, thus allowing you to use the `max()` function. While this may seem a minor benefit in this isolated example, in the context of a comprehensive library like the STL it allows the programmer to get extensive functionality for a new data type, just by defining a few operators for it. Merely defining < allows a type to be used with the standard `sort()`, `stable_sort()`, and `binary_search()` algorithms; data structures such as `set`s, heaps, and associative arrays; and more.

As a counterexample, the standard type `complex` does not define the < operator, because there is no strict order on COMPLEX NUMBER[5]s. Therefore `max(x, y)` will fail with a compile error if *x* and *y* are `complex` values. Likewise, other templates that rely on < cannot be applied to `complex` data. Unfortunately, compilers historically generate somewhat esoteric and unhelpful error messages for this sort of error. Ensuring that a certain object adheres to a METHOD PROTOCOL[6] can alleviate this issue.

`{TYPEPARAMETER}` is just the arbitrary **TYPEPARAMETER** name that you want to use in your function. Some programmers prefer using just `T` in place of `TYPEPARAMETER`.

Let us say you want to create a swap function that can handle more than one data type... something that looks like this:

```
template <class SOMETYPE>
void swap (SOMETYPE &x, SOMETYPE &y)
{
  SOMETYPE temp = x;
  x = y;
  y = temp;
}
```

The function you see above looks really similar to any other swap function, with the differences being the template <class SOMETYPE> line before the function definition and the instances of SOMETYPE in the code. Everywhere you would normally need to have the name or class of the datatype that you're using, you now replace with the arbitrary name that you used in the template <class SOMETYPE>. For example, if you had **SUPERDUPERTYPE** instead of **SOMETYPE**, the code would look something like this:

5 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPLEX%20NUMBER

6 HTTP://EN.WIKIPEDIA.ORG/WIKI/PROTOCOL%20%28COMPUTER%20SCIENCE%29

```
template <class SUPERDUPERTYPE>
void swap (SUPERDUPERTYPE &x, SUPERDUPERTYPE &y)
{
  SUPERDUPERTYPE temp = x;
  x = y;
  y = temp;
}
```

As you can see, you can use whatever label you wish for the template **TYPEPARAMETER**, as long as it is not a reserved word.

5.1.2 Class template

A *class template* extends the same concept to classes. Class templates are often used to make generic containers. For example, the STL has a LINKED LIST[7] container. To make a linked list of integers, one writes `list<int>`. A list of strings is denoted `list<string>`. A `list` has a set of standard functions associated with it, which work no matter what you put between the brackets.

If you want to have more than one template **TYPEPARAMETER**, then the syntax would be:

```
template <class SOMETYPE1, class SOMETYPE2, ...>
```

Templates and Classes

Let us say that rather than create a simple templated function, you would like to use templates for a class, so that the class may handle more than one datatype. You may have noticed that some classes are able to accept a type as a parameter and create variations of an object based on that type (for example the classes of the STL container class hierarchy). This is because they are declared as templates using syntax not unlike the one presented below:

```
template <class T> class Foo
{
public:
  Foo();
  void some_function();
  T some_other_function();

private:
  int member_variable;
```

7 HTTP://EN.WIKIPEDIA.ORG/WIKI/LINKED%20LIST

```
  T parametrized_variable;
};
```

Defining member functions of a template class is somewhat like defining a function template, except for the fact, that you use the scope resolution operator to indicate that this is the template classes' member function. The one important and non-obvious detail is the requirement of using the template operator containing the parametrized type name after the class name.

The following example describes the required syntax by defining functions from the example class above.

```
template <class T> Foo<T>::Foo()
{
  member_variable = 0;
}

template <class T> void Foo<T>::some_function()
{
  cout << "member_variable = " << member_variable << endl;
}

template <class T> T Foo<T>::some_other_function()
{
  return parametrized_variable;
}
```

As you may have noticed, if you want to declare a function that will return an object of the parametrized type, you just have to use the name of that parameter as the function's return type.

Note:
A class template can be used to avoid the overhead of virtual member functions in inheritance. Since the type of class is known at compile-time, the class template will not need the virtual pointer table that is required by a class with virtual member functions. This distinction also permits the inlining of the function members of a class template.

5.1.3 Advantages and disadvantages

Some uses of templates, such as the `max()` function, were previously filled by function-like PREPROCESSOR[8] MACRO[9]s.

8 HTTP://EN.WIKIPEDIA.ORG/WIKI/PREPROCESSOR
9 HTTP://EN.WIKIPEDIA.ORG/WIKI/MACRO

```
// a max() macro
#define max(a,b)    ((a) < (b) ? (b) : (a))
```

Both macros and templates are expanded at compile time. Macros are always expanded inline; templates can also be expanded as inline functions when the compiler deems it appropriate. Thus both function-like macros and function templates have no run-time overhead.

However, templates are generally considered an improvement over macros for these purposes. Templates are type-safe. Templates avoid some of the common errors found in code that makes heavy use of function-like macros. Perhaps most importantly, templates were designed to be applicable to much larger problems than macros. The definition of a function-like macro must fit on a single logical line of code.

There are three primary drawbacks to the use of templates. First, many compilers historically have very poor support for templates, so the use of templates can make code somewhat less portable. Second, almost all compilers produce confusing, unhelpful error messages when errors are detected in template code. This can make templates difficult to develop. Third, each use of a template may cause the compiler to generate extra code (an *instantiation* of the template), so the indiscriminate use of templates can lead to CODE BLOAT[10], resulting in excessively large executables.

The other big disadvantage of templates is that to replace a #define like max which acts identically with dissimilar types or function calls is impossible. Templates have replaced using #defines for complex functions but not for simple stuff like max(a,b). For a full discussion on trying to create a template for the #define max, see the paper "MIN, MAX AND MORE"[11] that Scott Meyer wrote for *C++ Report* in January 1995.

The biggest advantage of using templates, is that a complex algorithm can have a simple interface that the compiler then uses to choose the correct implementation based on the type of the arguments. For instance, a searching algorithm can take advantage of the properties of the container being searched. This technique is used throughout the C++ standard library.

10 HTTP://EN.WIKIPEDIA.ORG/WIKI/CODE%20BLOAT

11 HTTP://WWW.ARISTEIA.COM/PAPERS/C%2B%2BREPORTCOLUMNS/JAN95.PDF

5.1.4 Linkage problems

While linking a template-based program consisting over several modules spread over a couple files, it is a frequent and mystifying situation to find that the object code of the modules won't link due to 'unresolved reference to (insert template member function name here) in (...)'. The offending function's implementation is there, so why is it missing from the object code? Let us stop a moment and consider how can this be possible.

Assume you have created a template based class called Foo and put its declaration in the file Util.hpp along with some other regular class called Bar:

```
template <class T> Foo
{
public:
  Foo();
  T some_function();
  T some_other_function();
  T some_yet_other_function();
  T member;
};

class Bar
{
  Bar();
  void do_something();
};
```

Now, to adhere to all the rules of the art, you create a file called Util.cc, where you put all the function definitions, template or otherwise:

```
#include "Util.hpp"

template <class T> T Foo<T>::some_function()
{
 ...
}

template <class T> T Foo<T>::some_other_function()
{
 ...
}

template <class T> T Foo<T>::some_yet_other_function()
{
 ...
}
```

and, finally:

```
void Bar::do_something()
```

```
{
  Foo<int> my_foo;
  int x = my_foo.some_function();
  int y = my_foo.some_other_function();
}
```

Next, you compile the module, there are no errors, you are happy. But suppose there is an another (main) module in the program, which resides in MyProg.cc:

```
#include "Util.hpp"       // imports our utility classes' declarations, including
the template

int main()
{
  Foo<int> main_foo;
  int z = main_foo.some_yet_other_function();
  return 0;
}
```

This also compiles clean to the object code. Yet when you try to link the two modules together, you get an error saying there is an undefined reference to Foo<int>::some_yet_other function() in MyProg.cc. You defined the template member function correctly, so what is the problem?

As you remember, templates are instantiated at compile-time. This helps avoid code bloat, which would be the result of generating all the template class and function variants for all possible types as its parameters. So, when the compiler processed the Util.cc code, it saw that the only variant of the Foo class was Foo<int>, and the only needed functions were:

```
int Foo<int>::some_function();
int Foo<int>::some_other_function();
```

No code in Util.cc required any other variants of Foo or its methods to exist, so the compiler generated no code other than that. There is no implementation of some_yet_other_function() in the object code, just as there is no implementation for

```
double Foo<double>::some_function();
```

or

```
string Foo<string>::some_function();
```

The MyProg.cc code compiled without errors, because the member function of Foo it uses is correctly declared in the Util.hpp header, and it is expected that it will be available upon linking. But it is not and hence the error, and a lot of nuisance if

you are new to templates and start looking for errors in your code, which ironically is perfectly correct.

The solution is somewhat compiler dependent. For the GNU compiler, try experimenting with the -frepo flag, and also reading the template-related section of 'info gcc' (node "Template Instantiation": "Where is the Template?") may prove enlightening. In Borland, supposedly, there is a selection in the linker options, which activates 'smart' templates just for this kind of problem.

The other thing you may try is called explicit instantiation. What you do is create some dummy code in the module with the templates, which creates all variants of the template class and calls all variants of its member functions, which you know are needed elsewhere. Obviously, this requires you to know a lot about what variants you need throughout your code. In our simple example this would go like this:

1. Add the following class declaration to Util.hpp:

```
class Instantiations
{
private:
  void Instantiate();
};
```

2. Add the following member function definition to Util.cc:

```
void Instantiations::Instantiate()
{
  Foo<int> my_foo;
  my_foo.some_yet_other_function();
  // other explicit instantiations may follow
}
```

Of course, you never need to actual instantiate the Instantiations class, or call any of its methods. The fact that they just exist in the code makes the compiler generate all the template variations which are required. Now the object code will link without problems.

There is still one, if not elegant, solution. Just move all the template functions' definition code to the Util.hpp header file. This is not pretty, because header files are for declarations, and the implementation is supposed to be defined elsewhere, but it does the trick in this situation. While compiling the MyProg.cc (and any other modules which include Util.hpp) code, the compiler will generate all the template variants which are needed, because the definitions are readily available.

12

5.1.5 Template Meta-programming Overview

Template meta-programming (TMP) refers to uses of the C++ template system to perform computation at compile-time within the code. It can, for the most part, be considered to be "programming with types" — in that, largely, the "values" that TMP works with are specific C++ types. Using types as the basic objects of calculation allows the full power of the type-inference rules to be used for general-purpose computing.

Compile-time programming

The preprocessor allows certain calculations to be carried out at compile time, meaning that by the time the code has finished compiling the decision has already been taken, and can be left out of the compiled executable. The following is a very contrived example:

```
#define myvar 17

#if myvar % 2
   cout << "Constant is odd" << endl;
#else
   cout << "Constant is even" << endl;
#endif
```

This kind of construction does not have much application beyond conditional inclusion of platform-specific code. In particular there's no way to iterate, so it can not be used for general computing. Compile-time programming with templates works in a similar way but is much more powerful, indeed it is actually *Turing complete*.

Traits classes are a familiar example of a simple form of template meta-programming: given input of a type, they compute as output properties associated with that type (for example, std::iterator_traits<> takes an iterator type as input, and computes properties such as the iterator's difference_type, value_type and so on).

12 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

The nature of template meta-programming

Template meta-programming is much closer to functional programming than ordinary idiomatic C++ is. This is because 'variables' are all immutable, and hence it is necessary to use recursion rather than iteration to process elements of a set. This adds another layer of challenge for C++ programmers learning TMP: as well as learning the mechanics of it, they must learn to think in a different way.

Limitations of Template Meta-programming

Because template meta-programming evolved from an unintended use of the template system, it is frequently cumbersome. Often it is very hard to make the intent of the code clear to a maintainer, since the natural meaning of the code being used is very different from the purpose to which it is being put. The most effective way to deal with this is through reliance on idiom; if you want to be a productive template meta-programmer you will have to learn to recognize the common idioms.

It also challenges the capabilities of older compilers; generally speaking, compilers from around the year 2000 and later are able to deal with much practical TMP code. Even when the compiler supports it, the compile times can be extremely large and in the case of a compile failure the error messages are frequently impenetrable.

Some coding standards go as far as to outlaw template meta-programming, at least outside of third-party libraries like Boost.

History of TMP

Historically TMP is something of an accident; it was discovered during the process of standardizing the C++ language that its template system happens to be **Turing-complete**, i.e., capable in principle of computing anything that is computable. The first concrete demonstration of this was a program written by Erwin Unruh which computed prime numbers *although it did not actually finish compiling*: the list of prime numbers was part of an error message generated by the compiler on attempting to compile the code.HTTP://ASZT.INF.ELTE.HU/~GSD/HALADO_-CPP/CH06S04.HTML#STATIC-METAPROGRAMMING[13] TMP has since advanced considerably, and is now a practical tool for library builders in C++, though its

13 HTTP://ASZT.INF.ELTE.HU/~{}GSD/HALADO_CPP/CH06S04.HTML#STATIC-METAPROGRAMMING

complexities mean that it is not generally appropriate for the majority of applications or systems programming contexts.

```
#include <iostream>

template <int p, int i>
class is_prime {
public:
        enum { prim = (p==2) || (p%i) && is_prime<(i>2?p:0),i-1>::prim
        };
};

template<>
class is_prime<0,0> {
public:
        enum {prim=1};
};

template<>
class is_prime<0,1> {
public:
        enum {prim=1};
};

template <int i>
class Prime_print {      // primary template for loop to print prime numbers
public:
        Prime_print<i-1> a;
        enum { prim = is_prime<i,i-1>::prim
        };
        void f() {
                a.f();
                if (prim)
                {
                        std::cout << "prime number:" << i << std::endl;
                }
        }
};

template<>
class Prime_print<1> {   // full specialization to end the loop
public:
        enum {prim=0};
        void f() {
        };
};


#ifndef LAST
#define LAST 18
#endif

int main()
{
   Prime_print<LAST> a;
   a.f();
```

```
}
```

Building Blocks

Values

The 'variables' in TMP are not really variables since their values can not be altered, but you can have named values that you use rather like you would variables in ordinary programming. When programming with types, named values are typedefs:

```
struct ValueHolder
{
   typedef int value;
};
```

You can think of this as 'storing' the `int` type so that it can be accessed under the `value` name. Integer values are usually stored as members in an enum:

```
struct ValueHolder
{
   enum { value = 2 };
};
```

This again stores the value so that it can be accessed under the name `value`. Neither of these examples is any use on its own, but they form the basis of most other TMP, so they are vital patterns to be aware of.

Functions

A function maps one or more input parameters into an output value. The TMP analogue to this is a template class:

```
template<int X, int Y>
struct Adder
{
   enum { result = X + Y };
};
```

This is a function that adds its two parameters and stores the result in the `result enum` member. You can call this at compile time with something like `Adder<1, 2>::result`, which will be expanded at compile time and act exactly like a literal `3` in your program.

Branching

A conditional branch can be constructed by writing two alternative specialisations of a template class. The compiler will choose the one that fits the types provided, and a value defined in the instantiated class can then be accessed. For example, consider the following partial specialisation:

```
template<typename X, typename Y>
struct SameType
{
   enum { result = 0 };
};

template<typename T>
struct SameType<T, T>
{
   enum { result = 1 };
};
```

This tells us if the two types it is instantiated with are the same. This might not seem very useful, but it can see through typedefs that might otherwise obscure whether types are the same, and it can be used on template arguments in template code. You can use it like this:

```
if (SameType<SomeThirdPartyType, int>::result)
{
   // ... Use some optimised code that can assume the type is an int
}
else
{
   // ... Use defensive code that doesn't make any assumptions about the type
}
```

The above code isn't very idiomatic: since the types can be identified at compile-time, the `if()` block will always have a trivial condition (it'll always resolve to either `if (1) { ... }` or `if (0) { ... }`). However, this does illustrate the kind of thing that can be achieved.

Recursion

Since you don't have mutable variables available when you're programming with templates, it's impossible to iterate over a sequence of values. Tasks that might be achieved with iteration in standard C++ have to be redefined in terms of recursion, i.e. a function that calls itself. This usually takes the shape of a template class whose output value recursively refers to itself, and one or more specialisations that give fixed values to prevent infinite recursion. You can think of this as a combination of the function and conditional branch ideas described above.

Calculating factorials is naturally done recursively: $0! = 1$, and for $n > 0$, $n! = n * (n-1)!$. In TMP, this corresponds to a class template "factorial" whose general form uses the recurrence relation, and a specialization of which terminates the recursion.

First, the general (unspecialized) template says that `factorial<n>::value` is given by `n*factorial<n-1>::value`:

```
template <unsigned n>
struct factorial
{
  enum { value = n * factorial<n-1>::value };
};
```

Next, the specialization for zero says that `factorial<0>::value` evaluates to 1:

```
template <>
struct factorial<0>
{
  enum { value = 1 };
};
```

And now some code that "calls" the factorial template at compile-time:

```
int main() {
  // Because calculations are done at compile-time, they can be
  // used for things such as array sizes.
  int array[ factorial<7>::value ];
}
```

Observe that the `factorial<N>::value` member is expressed in terms of the `factorial<N>` template, but this can't continue infinitely: each time it is evaluated, it calls itself with a progressively smaller (but non-negative) number. This must eventually hit zero, at which point the specialisation kicks in and evaluation doesn't recurse any further.

Example: Compile-time "If"

The following code defines a meta-function called "if_"; this is a class template that can be used to choose between two types based on a compile-time constant, as demonstrated in **main** below:

```
template <bool Condition, typename TrueResult, typename FalseResult>
class if_;

template <typename TrueResult, typename FalseResult>
struct if_<true, TrueResult, FalseResult>
```

```
{
  typedef TrueResult result;
};

template <typename TrueResult, typename FalseResult>
struct if_<false, TrueResult, FalseResult>
{
  typedef FalseResult result;
};

int main()
{
  typename if_<true, int, void*>::result number(3);
  typename if_<false, int, void*>::result pointer(&number);

   typedef typename if_<(sizeof(void *) > sizeof(uint32_t)), uint64_t,
 uint32_t>::result
      integral_ptr_t;

   integral_ptr_t converted_pointer = reinterpret_cast<integral_ptr_t>(pointer);
}
```

On line 18, we evaluate the `if_` template with a true value, so the type used is the first of the provided values. Thus the entire expression `if_<true, int, void*>::result` evaluates to `int`. Similarly, on line 19 the template code evaluates to `void *`. These expressions act exactly the same as if the types had been written as literal values in the source code.

Line 21 is where it starts to get clever: we define a type that depends on the value of a platform-dependent `sizeof` expression. On platforms where pointers are either 32 or 64 bits, this will choose the correct type at compile time without any modification, and without preprocessor macros. Once the type has been chosen, it can then be used like any other type.

Note:
This code is just an illustration of the power of template meta-programming, it is not meant to illustrate good cross-platform practice with pointers.

For comparison, this problem is best attacked in C90 as follows

```
# include <stddef.h>
typedef size_t integral_ptr_t;
typedef int the_correct_size_was_chosen [sizeof (integral_ptr_t) >= sizeof (void
 *)? 1: -1];
```

As it happens, the library-defined type `size_t` should be the correct choice for this particular problem on any platform. To ensure this, line 3 is used as a compile time check to see if the selected type is actually large enough; if not, the array type `the_correct_size_was_chosen` will be defined with a negative length, causing

a compile-time error. In C99, `<stdint.h>` may define the types `intptr_h` and `uintptr_h`.

Conventions for "Structured" TMP

14

5.2 Standard Template Library (STL)

The **Standard Template Library** (**STL**), part of the C++ STANDARD LIBRARY[15], offers collections of algorithms, containers, iterators, and other fundamental components, implemented as templates, classes, and functions essential to extend functionality and standardization to C++. STL main focus is to provide improvements implementation standardization with emphasis in performance and correctness.

Instead of wondering if your array would ever need to hold 257 records or having nightmares of string buffer overflows, you can enjoy **vector** and **string** that automatically extend to contain more records or characters. For example, vector is just like an array, except that vector's size can expand to hold more cells or shrink when fewer will suffice. One must keep in mind that the STL does not conflict with OOP but in itself is not object oriented; In particular it makes no use of runtime polymorphism (i.e., has no virtual functions).

The true power of the STL lies not in its CONTAINER[16] classes, but in the fact that it is a framework, combining algorithms with data structures using indirection through iterators to allow generic implementations of higher order algorithms to work efficiently on varied forms of data. To give a simple example, the same **std::copy** function can be used to copy elements from one array to another, or to copy the bytes of a file, or to copy the whitespace-separated words in "text like this" into a container such as **std::vector<std::string>**.

```
// std::copy from array a to array b
int a[10] = { 3,1,4,1,5,9,2,6,5,4 };
int b[10];
std::copy(&a[0], &a[10], b);
```

14 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
15 Chapter 3.1.2 on page 45
16 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23CONTAINERS

```
// std::copy from input stream a to an arbitrary OutputIterator
template <typename OutputIterator>
void f(std::istream &a, OutputIterator destination) {
  std::copy(std::istreambuf_iterator<char>(a),
            std::istreambuf_iterator<char>(),
            destination);
}

// std::copy from a buffer containing text, inserting items in
// order at the back of the container called words.
std::istringstream buffer("text like this");
std::vector<std::string> words;
std::copy(std::istream_iterator<std::string>(buffer),
          std::istream_iterator<std::string>(),
          std::back_inserter(words));
assert(words[0] == "text");
assert(words[1] == "like");
assert(words[2] == "this");
```

5.2.1 History



Figure 25: Alexander Stepanov

The C++ Standard Library incorporated part of the STL (published as a software library by SGI[17]/Hewlett-Packard Company). The primary implementer of the C++ Standard Template Library was ALEXANDER STEPANOV[18].

Today we call STL to what was adopted into the C++ Standard. The ISO C++ does not specify header content, and allows implementation of the STL either in the headers, or in a true library.

17 HTTP://EN.WIKIPEDIA.ORG/WIKI/SILICON%20GRAPHICS
18 HTTP://EN.WIKIPEDIA.ORG/WIKI/ALEXANDER%20STEPANOV

Note:
In an interview Alexander Stepanov, stated that he originally, wanted all auxiliary functions in STL to be visible but it was not politically possible, especially the heap functions. That Bjarne did reduce the number of components in STL by a factor of two as to permit the adoption into the standard.

Compilers will already have one implementation included as part of the C++ Standard (i.e., MS Visual Studio uses the Dinkum STL). All implementations will have to comply to the standard's requirements regarding functionality and behavior, but consistency of programs across all major hardware implementations, operating systems, and compilers will also depends on the portability of the STL implementation. They may also offer extended features or be optimized to distinct setups.

List of STL implementations.

- libstdc++ from gnu (was part of libg++)
- SGI STL library (HTTP://WWW.SGI.COM/TECH/STL/)[19] free STL implementation.
- Rogue Wave standard library (HP, SGI, SunSoft, Siemens-Nixdorf) / APACHE C++ STANDARD LIBRARY (STDCXX)[20]
- Dinkum STL library by P.J. Plauger (HTTP://WWW.DINKUMWARE.COM/)[21] commercial STL implementation widely used, since it was licensed in is co-maintained by Microsoft and it is the STL implementation that ships with Visual Studio.

There are many different implementations of the STL, all based on the language standard but nevertheless differing from each other, making it transparent for the programmer, enabling specialization and rapid evolution of the code base.

Open Source versions of the STL are available (can be useful to consult)

- Apache C++ Standard Library (STDCXX) (HTTP://STDCXX.APACHE.ORG/[22]).

19 HTTP://WWW.SGI.COM/TECH/STL/)
20 HTTP://EN.WIKIPEDIA.ORG/WIKI/APACHE%20C%2B%2B%20STANDARD%20LIBRARY
21 HTTP://WWW.DINKUMWARE.COM/)
22 HTTP://STDCXX.APACHE.ORG/

- STLport STL library (HTTP://WWW.STLPORT.COM/)[23] free and highly cross-platform implementation based on the SGI implementation.

Note:
There are advantages on having compartmentalized functionalities, some developers actively avoid using some of the language features, for a multitude of reasons. C++ permits the programmer to chose how to express himself, have control over the development paradigms and not be constricted by an higher level of abstraction.

5.2.2 Containers

The containers we will discuss in this section of the book are part of the standard namespace (std::). They all originated in the original SGI implementation of the STL.

Note:
When choosing a container, you should have in mind what makes them different, this will help you produce more efficient code. See also the OPTIMIZATION SECTION[a] of the book, about USING THE RIGHT DATA IN THE RIGHT CONTAINER[b].

a Chapter 6.7.2 on page 628
b Chapter 6.8.1 on page 630

Sequence Containers

Sequences - easier than arrays

Sequences are similar to C arrays, but they are easier to use. Vector is usually the first sequence to be learned. Other sequences, list and double-ended queues, are similar to vector but more efficient in some special cases. (Their behavior is also different in important ways concerning validity of iterators when the container is changed; iterator validity is an important, though somewhat advanced, concept when using containers in C++.)

23 HTTP://WWW.STLPORT.COM/)

- vector - "an easy-to-use array"
- list - in effect, a doubly-linked list
- deque - double-ended queue (properly pronounced "deck", often mispronounced as "dee-queue")

vector

The **vector** is a template class in itself, it is a *Sequence Container* and allows you to easily create a DYNAMIC ARRAY[24] of elements (one type per instance) of almost any data-type or object within a programs when using it. The **vector** class handles most of the memory management for you.

Since a **vector** contain contiguous elements it is an ideal choice to replace the old C style array, in a situation where you need to store data, and ideal in a situation where you need to store dynamic data as an array that changes in size during the program's execution (old C style arrays can't do it). However, vectors do incur a very small overhead compared to static arrays (depending on the quality of your compiler), and cannot be initialized through an initialization list.

Note:
Vector is known to be slow when using the MSVC compiler due to the SECURE_SCL flag, that, by default, forces bounds checking even in optimized builds.

Accessing members of a vector or appending elements takes a fixed amount of time, no matter how large the vector is, whereas locating a specific value in a vector element or inserting elements into the vector takes an amount of time directly proportional to its location in it (size dependent).

24 HTTP://EN.WIKIPEDIA.ORG/WIKI/DYNAMIC%20ARRAY

Note:
If you create a vector you can access its data using consecutive pointers:

```
std::vector<type> myvector(8); type * ptr = myvector[0]; ptr[0],
ptr[7]; // access the first and last objects in myvector
```

this information is present in INCITS/ISO/IEC 14882-2003 but was not properly documented in the 1998 version of the C++ standard.

Be aware that ptr[i] is faster than myvector.at(i) because no error checking is performed. Watch out for how long that pointer is valid. The contiguous nature of vectors is most often important when interfacing to C code.
You should also keep in mind that std::vector<T>::iterator may not be a pointer; using an iterator is the safest mode to access a container but safety has always a cost in performance.

Example

```
/*
David Cary 2009-03-04
quick demo for wikibooks
*/

#include <iostream>
#include <vector>
using namespace std;

vector<int> pick_vector_with_biggest_fifth_element(vector<int> left,vector<int>
 right)
{
    if(left[5] < right[5])
    {
        return( right );
    }
    // else
    return left ;
}

int* pick_array_with_biggest_fifth_element(int * left,int * right)
{
    if(left[5] < right[5])
    {
        return( right );
    }
    // else
    return left ;
}
```

```
int vector_demo(void)
{
    cout << "vector demo" << endl;
    vector<int> left(7);
    vector<int> right(7);

    left[5] = 7;
    right[5] = 8;
    cout << left[5] << endl;
    cout << right[5] << endl;
    vector<int> biggest(pick_vector_with_biggest_fifth_element( left, right ) );
    cout << biggest[5] << endl;

    return 0;
}

int array_demo(void)
{
    cout << "array demo" << endl;
    int left[7];
    int right[7];

    left[5] = 7;
    right[5] = 8;
    cout << left[5] << endl;
    cout << right[5] << endl;
    int * biggest =
        pick_array_with_biggest_fifth_element( left, right );
    cout << biggest[5] << endl;

    return 0;
}

int main(void)
{
    vector_demo();
    array_demo();
}
```

Member Functions

The `vector` class models the CONTAINER[25] CONCEPT[26], which means it has `begin()`, `end()`, `size()`, `max_size()`, `empty()`, and `swap()` methods.

25 HTTP://WWW.SGI.COM/TECH/STL/CONTAINER.HTML

26 HTTP://EN.WIKIPEDIA.ORG/WIKI/CONCEPT%20%28GENERIC%20PROGRAMMING%29

Note:
Since most vector (or deque) implementations typically reserves some extra internal storage for future growth. Prefer the `swap()` method when altering a standard vector size (or freeing the memory used) when memory resources becomes a factor.

- informative
  - `vector::front` - Returns reference to first element of vector.
  - `vector::back` - Returns reference to last element of vector.
  - `vector::size` - Returns number of elements in the vector.
  - `vector::empty` - Returns true if vector has no elements.
- standard operations
  - `vector::insert` - Inserts elements into a vector (single & range), shifts later elements up. Inefficient.
  - `vector::push_back` - Appends (inserts) an element to the end of a vector, allocating memory for it if necessary. AMORTIZED[27] O(1) time.
  - `vector::erase` - Deletes elements from a vector (single & range), shifts later elements down. Inefficient.
  - `vector::pop_back` - Erases the last element of the vector, (possibly reducing capacity - usually it isn't reduced, but this depends on particular STL implementation). AMORTIZED[28] O(1) time.
  - `vector::clear` - Erases all of the elements. Note however that if the data elements are pointers to memory that was created dynamically (e.g., the **new** operator was used), the memory will not be freed.
- allocation/size modification
  - `vector::assign` - Used to delete a *origin* **vector** and copies the specified elements to an empty *target* **vector**.
  - `vector::reserve` - Changes capacity (allocates more memory) of vector, if needed. In many STL implementations capacity can only grow, and is never reduced.
  - `vector::capacity` - Returns current capacity (allocated memory) of vector.
  - `vector::resize` - Changes the vector size.
- iteration
  - `vector::begin` - Returns an iterator to start traversal of the vector.
  - `vector::end` - Returns an iterator that points just beyond the end of the vector.
  - `vector::at` - Returns a reference to the data element at the specified location in the **vector**, with bounds checking.

27 HTTP://EN.WIKIPEDIA.ORG/WIKI/AMORTIZED%20ANALYSIS
28 HTTP://EN.WIKIPEDIA.ORG/WIKI/AMORTIZED%20ANALYSIS

Note:
It is important to remember the distinctions of capacity(), size() and empty() when dealing with containers.

```
vector<int> v;
for (vector<int>::iterator it = v.begin(); it!=v.end(); ++it/* increment operand
 is used to move to next element*/) {
    cout << *it << endl;
}
```

vector::Iterators

std::vector<T> provides Random Access Iterators; as with all containers, the primary access to iterators is via begin() and end() member functions. These are overloaded for const- and non-const containers, returning iterators of types std::vector<T>::const_iterator and std::vector<T>::iterator respectively.

vector examples

```
/* Vector sort example */
#include <iostream>
#include <vector>

int main()
{
        using namespace std;

        cout << "Sorting STL vector, \"the easier array\"... " << endl;
        cout << "Enter numbers, one per line.  Press ctrl-D to quit." << endl;

        vector<int> vec;
        int tmp;
        while (cin>>tmp) {
                vec.push_back(tmp);
        }

        cout << "Sorted: " << endl;
        sort(vec.begin(), vec.end());
        int i = 0;
        for (i=0; i<vec.size(); i++) {
                cout << vec[i] << endl;;
        }

        return 0;
}
```

The call to `sort` above actually calls an instantiation of the function template `std::sort`, which will work on any half-open range specified by two random access iterators.

If you like to make the code above more "STLish" you can write this program in the following way:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

int main()
{
	using namespace std;

	cout << "Sorting STL vector, \"the easier array\"... " << endl;
	cout << "Enter numbers, one per line.  Press ctrl-D to quit." << endl;

	vector<int> vec(istream_iterator<int>(cin), istream_iterator<int>());

	sort(vec.begin(), vec.end());

	cout << "Sorted: " << endl;

	copy(vec.begin(), vec.end(), ostream_iterator<int>(cout, "\n"));

	return 0;
}
```

Linked lists

The STL provides a class template called **list** (part of the standard namespace (std::)) which implements a non-intrusive doubly-LINKED LIST[29]. Linked lists can insert or remove elements in the middle in constant time, but do not have random access. One useful feature of **std::list** is that references, pointers and iterators to items inserted into a list remain valid so long as that item remains in the list.

Note:
Consider using vector instead of list for better cache coherency and avoid "death by swapping", see the OPTIMIZATION SECTION[a], about using the RIGHT DATA IN THE RIGHT CONTAINER[b].

a Chapter 6.7.2 on page 628
b Chapter 6.8.1 on page 630

29 HTTP://EN.WIKIPEDIA.ORG/WIKI/LINKED%20LIST

list examples

Associative Containers (key and value)

This type of container point to each element in the container with a key value, thus simplifying searching containers for the programmer. Instead of iterating through an array or vector element by element to find a specific one, you can simply ask for people["tero"]. Just like vectors and other containers, associative containers can expand to hold any number of elements.

Maps and Multimaps

map and *multimap* are associative containers that manage key/value pairs as elements as seen above. The elements of each container will sort automatically using the actual key for sorting criterion. The difference between the two is that maps do not allow duplicates, whereas, multimaps does.

- map - unique keys
- multimap - same key can be used many times
- set - unique key is the value
- multiset - key is the value, same key can be used many times

```
/* Map example - character distribution  */
#include <iostream>
#include <map>
#include <string>
#include <cctype>

using namespace std;

int main()
{
        /* Character counts are stored in a map, so that
           * character is the key.
           * Count of char a is chars['a']. */
        map<char, long> chars;

        cout << "chardist - Count character distributions" << endl;
        cout << "Type some text. Press ctrl-D to quit." << endl;
        char c;
        while (cin.get(c)) {
                // Upper A and lower a are considered the same
                c=tolower(static_cast<unsigned char>(c));
                chars[c]=chars[c]+1; // Could be written as ++chars[c];
        }

        cout << "Character distribution: " << endl;
```

```
        string alphabet("abcdefghijklmnopqrstuvwxyz");
        for (string::iterator letter_index=alphabet.begin(); letter_index !=
alphabet.end(); letter_index++) {
                if (chars[*letter_index] != 0) {
                        cout << char(toupper(*letter_index))
                             << ":" << chars[*letter_index]
                             << "\t" << endl;
                }
        }
        return 0;
}
```

Container Adapters

- stack - last in, first out (LIFO)
- queue - first in, first out (FIFO)
- priority queue

5.2.3 Iterators

C++'s iterators are one of the foundation of the STL. Iterators exist in languages other than C++, but C++ uses an unusual form of iterators, with pros and cons.

In C++, an iterator is a *concept* rather than a specific type, they are a generalization of the pointers as an abstraction for the use of containers. Iterators are further divided based on properties such as traversal properties.

The basic idea of an iterator is to provide a way to navigate over some collection of objects concept.

Some (overlapping) categories of iterators are:

- Singular iterators
- Invalid iterators
- Random access iterators
- Bidirectional iterators
- Forward iterators
- Input iterators
- Output iterators
- Mutable iterators

A pair of iterators **[begin, end)** is used to define a HALF OPEN RANGE[30], which includes the element identified from **begin** to **end**, except for the element identified by **end**. As a special case, the half open range **[x, x)** is empty, for any valid iterator x.

Note:
The range notation may vary, the meaning is to express the inclusion or exclusion of the range limits. An also common notation is [begin, end[(meaning begin is part of the range and end is not).

The most primitive examples of iterators in C++ (and likely the inspiration for their syntax) are the built-in pointers, which are commonly used to iterate over elements within arrays.

Iteration over a Container

Accessing (but not modifying) each element of a container `group` of type *C*<*T*> using an iterator.

```
for (
     typename C<T>::const_iterator iter = group.begin();
     iter != group.end();
     ++iter
    )
{
    T const &element = *iter;

    // access element here
}
```

Note the usage of typename. It informs the compiler that 'const_iterator' is a type as opposed to a static member variable. (It is only necessary inside templated code, and indeed in C++98 is invalid in regular, non-template, code. This may change in the next revision of the C++ standard so that the **typename** above is always permitted.)

Modifying each element of a container `group` of type *C*<*T*> using an iterator.

```
for (
     typename C<T>::iterator iter = group.begin();
     iter != group.end();
     ++iter
    )
```

30 HTTP://EN.WIKIBOOKS.ORG/WIKI/ALGEBRA%2FINTERVAL%20NOTATION

```
{
    T &element = *iter;

    // modify element here
}
```

When modifying the container itself while iterating over it, some containers (such as vector) require care that the iterator doesn't become invalidated, and end up pointing to an invalid element. For example, instead of:

```
for (i = v.begin(); i != v.end(); ++i) {
  ...
  if (erase_required) {
    v.erase(i);
  }
}
```

Do:

```
for (i = v.begin(); i != v.end(); ) {
  ...
  if (erase_required) {
      i = v.erase(i);
  } else {
      ++i;
  }
}
```

The `erase()` member function returns the next valid iterator, or `end()`, thus ending the loop. Note that `++i` is *not* executed when `erase()` has been called on an element.

5.2.4 Functors

A functor or function object, is an object that has an `operator ()`. The importance of functors is that they can be used in many contexts in which C++ functions can be used, whilst also having the ability to maintain state information. Next to iterators, functors are one of the most fundamental ideas exploited by the STL.

The STL provides a number of pre-built functor classes; std::less, for example, is often used to specify a default comparison function for algorithms that need to determine which of two objects comes "before" the other.

```
#include <vector>
#include <algorithm>
#include <iostream>
```

```
// Define the Functor for AccumulateSquareValues
template<typename T>
struct AccumulateSquareValues
{
    AccumulateSquareValues() : sumOfSquares()
    {
    }
    void operator()(const T& value)
    {
        sumOfSquares += value*value;
    }
    T Result() const
    {
        return sumOfSquares;
    }
    T sumOfSquares;
};

std::vector<int> intVec;
intVec.reserve(10);
for( int idx = 0; idx < 10; ++idx )
{
    intVec.push_back(idx);
}
AccumulateSquareValues<int> sumOfSquare = std::for_each(intVec.begin(),
                                                        intVec.end(),

AccumulateSquareValues<int>() );
std::cout << "The sum of squares for 1-10 is " << sumOfSquare.Result() <<
std::endl;

// note: this problem can be solved in another, more clear way:
// int sum_of_squares = std::inner_product(intVec.begin(), intVec.end(),
intVec.begin(), 0);
```

5.2.5 Algorithms

The STL also provides several useful algorithms, in the form of *template functions*, that are provided to, with the help of the iterator concept, manipulate the STL containers (or derivations).

The STL algorithms aren't restricted to STL containers, for instance:

```
#include <algorithm>

  int array[10] = { 2,3,4,5,6,7,1,9,8,0 }

  int* begin = &array[0];
  int* end = &array[0] + 10;

  std::sort( begin, end);// the sort algorithm will work on a C style array
```

The _if suffix

The _copy suffix

- Non-modifying algorithms
- Modifying algorithms
- Removing algorithms
- Mutating algorithms
- Sorting algorithms
- Sorted range algorithms
- Numeric algorithms

Permutations

Sorting and related operations

`sort`

`stable_sort`

`partial_sort`

Minimum and maximum

The standard library provides function templates `min` and `max`, which return the minimum and maximum of their two arguments respectively. Each has an overload available that allows you to customize the way the values are compared.

```
template<class T>
const T& min(const T& a, const T& b);

template<class T, class Compare>
const T& min(const T& a, const T& b, Compare c);

template<class T>
const T& max(const T& a, const T& b);

template<class T, class Compare>
const T& max(const T& a, const T& b, Compare c);
```

5.2.6 Allocators

Allocators are used by the Standard C++ Library (and particularly by the STL) to allow parameterization of memory allocation strategies.

The subject of allocators is somewhat obscure, and can safely be ignored by most application software developers. All standard library constructs that allow for specification of an allocator have a default allocator which is used if none is given by the user.

Custom allocators can be useful if the memory use of a piece of code is unusual in a way that leads to performance problems if used with the general-purpose default allocator. There are also other cases in which the default allocator is inappropriate, such as when using standard containers within an implementation of replacements for global operators new and delete.

31

5.3 Smart Pointers

Using raw pointers to store allocated data and then cleaning them up in the destructor can generally be considered a very bad idea since it is error-prone. Even temporarily storing allocated data in a raw pointer and then deleting it when done with it should be avoided for this reason. For example, if your code throws an exception, it can be cumbersome to properly catch the exception and delete all allocated objects.

Smart pointers can alleviate this headache by using the compiler and language semantics to ensure the pointer content is automatically released when the pointer itself goes out of scope.

```
#include <memory>
class A
{
public:
        virtual ~A() {}
        virtual char val() = 0;
};

class B : public A
{
public:
```

31 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

```
        virtual char val() { return 'B'; }
};

A* get_a_new_b()
{
        return new B();
}

bool some_func()
{
        bool rval = true;
        std::auto_ptr<A> a( get_a_new_b() );
        try {
                std::cout << a->val();
        } catch(...) {
                if( !a.get() ) {
                        throw "Memory allocation failure!";
                }
                rval = false;
        }
        return rval;
}
```

5.4 Semantics

The auto_ptr has semantics of strict ownership, meaning that the auto_ptr instance is the sole entity responsible for the object's lifetime. If an auto_ptr is copied, the source loses the reference. For example:

```
#include <iostream>
#include <memory>
using namespace std;

int main(int argc, char **arv)
{
    int *i = new int;
    auto_ptr<int> x(i);
    auto_ptr<int> y;

    y = x;

    cout << x.get() << endl;
    cout << y.get() << endl;
}
```

This code will print a NULL address for the first auto_ptr object and some non-NULL address for the second, showing that the source object lost the reference during the assignment (=). The raw pointer `i` in the example should not be deleted, as it will be deleted by the auto_ptr that owns the reference. In fact, `new int` could be passed directly into x, eliminating the need for `i`.

Notice that the object pointed by an auto_ptr is destructed using `operator delete`; this means that you should only use auto_ptr for pointers obtained with `operator new`. This excludes pointers returned by `malloc(), calloc() or realloc()` and `operator new[]`.

32

5.5 Exception Handling

EXCEPTION HANDLING[33] is a construct designed to handle the occurrence of exceptions, that is special conditions that changes the normal flow of program execution. Since when designing a programming task (a class or even a function), one cannot always assume that application/task will run or be completed correctly (exit with the result it was intended to). It may be the case that it will be just inappropriate for that given task to report an error message (return an error code) or just exit. To handle these types of cases, C++ supports the use of language constructs to separate error handling and reporting code from ordinary code, that is, constructs that can deal with these **exceptions** (errors and abnormalities) and so we call this global approach that adds uniformity to program design the **exception handling**.

An exception is said to be **thrown** at the place where some error or abnormal condition is detected. The throwing will cause the normal program flow to be aborted, in a **raised exception**. An exception is thrown programmatic, the programmer specifies the conditions of a throw.

In **handled exceptions**, execution of the program will resume at a designated block of code, called a **catch block**, which encloses the point of throwing in terms of program execution. The catch block can be, and usually is, located in a different function/method than the point of throwing. In this way, C++ supports *non-local error handling*. Along with altering the program flow, throwing of an exception passes an object to the catch block. This object can provide data that is necessary for the handling code to decide in which way it should react on the exception.

Consider this next code example of a **try** and **catch** block combination for clarification:

```
void AFunction()
{
    // This function does not return normally,
    // instead execution will resume at a catch block.
```

32 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
33 HTTP://EN.WIKIPEDIA.ORG/WIKI/EXCEPTION%20HANDLING

```
    // The thrown object is in this case of the type char const*,
    // i.e. it is a C-style string. More usually, exception
    // objects are of class type.
    throw "This is an exception!";
}

void AnotherFunction()
{
    // To catch exceptions, you first have to introduce
    // a try block via " try { ... } ". Then multiple catch
    // blocks can follow the try block.
    // " try { ... } catch(type 1) { ... } catch(type 2) { ... }"
    try
    {
        AFunction();
       // Because the function throws an exception,
       // the rest of the code in this block will not
       // be executed
    }
    catch(char const* pch)  // This catch block
                           // will react on exceptions
                           // of type char const*
    {
        // Execution will resume here.
        // You can handle the exception here.
    }
             // As can be seen
    catch(...) // The ellipsis indicates that this
             // block will catch exceptions of any type.
    {
       // In this example, this block will not be executed,
       // because the preceding catch block is chosen to
       // handle the exception.
    }
}
```

Unhandled exceptions on the other hand will result in a function termination and the STACK WILL BE UNWOUND[34] (stack allocated objects will have destructors called) as it looks for an exception handler. If none is found it will ultimately result in the termination of the program.

From the point of view of a programmer, raising an exception is a useful way to signal that a routine could not execute normally. For example, when an input argument is invalid (e.g. a zero denominator in division) or when a resource it relies on is unavailable (like a missing file, or a hard disk error). In systems without exceptions, routines would need to return some special error code. However, this is sometimes complicated by the SEMI-PREDICATE PROBLEM[35], in which users of the routine need to write extra code to distinguish normal return values from erroneous ones.

34 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23STACK%20UNWINDING
35 HTTP://EN.WIKIPEDIA.ORG/WIKI/SEMIPREDICATE%20PROBLEM

Because it is hard to write exception safe code, you should only use an exception when you have to - when an error has occurred that you can not handle. Do *not* use exceptions for the normal flow of the program. This example is *WRONG*.

```
void sum(int iA, int iB)
{
    throw iA + iB;
}

int main()
{
    int iResult;

    try
    {
        sum(2, 3);
    }
    catch(int iTmpResult)
    {
        // Here the exception is used instead of a return value!
        // This is  wrong!
        iResult = iTmpResult;
    }

    return 0;
}
```

5.5.1 Stack unwinding

Consider the following code

```
void g()
{
    throw std::exception();
}

void f()
{
    std::string str = "Hello"; // This string is newly allocated
    g();
}

int main()
{
    try
    {
        f();
    }
    catch(...)
    { }
}
```

The flow of the program:

- `main()` calls `f()`
- `f()` creates a local variable named str
- `str constructor allocates a memory chunk to hold the string` "Hello"
- `f()` calls `g()`
- `g()` throws an exception
- `f()` does not catch the exception.

Because the exception was not caught, we now need to exit `f()` in a clean fashion.

At this point, all the destructors of local variables previous to the throw

are called - This is called 'stack unwinding'.

- The destructor of str is called, which releases the memory occupied by it.

As you can see, the mechanism of 'stack unwinding' is essential to prevent resource leaks - without it, str would never be destroyed, and the memory it used would be lost forever.

- `main()` catches the exception
- The program continues.

The 'stack unwinding' guarantees destructors of local variables (stack variables) will be called when we leave its scope.

5.5.2 Throwing objects

There are several ways to throw an exception object.

Throw a pointer to the object:

```
void foo()
{
    throw new MyApplicationException();
}

void bar()
{
    try
    {
        foo();
    }
    catch(MyApplicationException* e)
    {
```

```
        // Handle exception
    }
}
```

But now, who is responsible to delete the exception? The handler? This makes code uglier. There must be a better way!

How about this:

```
void foo()
{
    throw MyApplicationException();
}

void bar()
{
    try
    {
        foo();
    }
    catch(MyApplicationException e)
    {
        // Handle exception
    }
}
```

Looks better! But now, the catch handler that catches the exception, does it by value, meaning that a copy constructor is called. This can cause the program to crash if the exception caught was a `bad_alloc` caused by insufficient memory. In such a situation, seemingly safe code that is assumed to handle memory allocation problems results in the program crashing with a failure of the exception handler. Moreover, catching by value may cause the copy to have different behavior because of object slicing.

The correct approach is:

```
void foo()
{
    throw MyApplicationException();
}

void bar()
{
    try
    {
        foo();
    }
    catch(MyApplicationException const& e)
    {
        // Handle exception
    }
}
```

This method has all the advantages - the compiler is responsible for destroying the object, and no copying is done at catch time!

The conclusion is that exceptions should be thrown by value, and caught by (usually const) reference.

5.5.3 Constructors and destructors

When an exception is thrown from a constructor, the object is not considered instantiated, and therefore its destructor will *not* be called. But all destructors of already successfully constructed base and member objects of the same master object will be called. Constructors of not yet constructed base or member objects of the same master object will not be executed. Example:

```
class A : public B, public C
{
public:
    D sD;
    E sE;
    A(void)
    :B(), C(), sD(), sE()
    {
    }
};
```

Let's assume the constructor of base class C throws. Then the order of execution is:

- B
- C (throws)
- ~B

Let's assume the constructor of member object sE throws. Then the order of execution is:

- B
- C
- sD
- sE (throws)
- ~sD
- ~C
- ~B

Thus if some constructor is executed, one can rely on that all other constructors of the same master object executed before, were successful. This enables one, to use an already constructed member or base object as an argument for the constructor of one of the following member or base objects of the same master object.

What happens when we allocate this object with `new`?

- Memory for the object is allocated
- The object's constructor throws an exception
  - The object was not instantiated due to the exception
- The memory occupied by the object is deleted
- The exception is propagated, until it is caught

The main purpose of throwing an exception from a constructor is to inform the program/user that the creation and initialization of the object did not finish correctly. This is a very clean way of providing this important information, as constructors do not return a separate value containing some error code (as an initialization function would).

In contrast, it is strongly recommended not to throw exceptions inside a destructor. It is important to note when a destructor is called:

- as part of a normal deallocation (exit from a scope, delete)
- as part of a stack unwinding that handles a previously thrown exception.

In the former case, throwing an exception inside a destructor can simply cause memory leaks due to incorrectly deallocated object. In the latter, the code must be more clever. If an exception was thrown as part of the stack unwinding caused by another exception, there is no way to choose which exception to handle first. This is interpreted as a failure of the exception handling mechanism and that causes the program to call the function terminate.

To address this problem, it is possible to test if the destructor was called as part of an exception handling process. To this end, one should use the standard library function uncaught_exception, which returns true if an exception has been thrown, but hasn't been caught yet. All code executed in such a situation must not throw another exception.

Situations where such careful coding is necessary are extremely rare. It is far safer and easier to debug if the code was written in such a way that destructors did not throw exceptions at all.

5.5.4 Writing exception safe code

Exception safety

A piece of code is said to be **exception-safe**, if run-time failures within the code will not produce ill effects, such as MEMORY LEAK[36]s, garbled stored data, or invalid output. Exception-safe code must satisfy INVARIANT[37]s placed on the code even if exceptions occur. There are several levels of exception safety:

1. **Failure transparency**, also known as the **no throw guarantee**: Operations are guaranteed to succeed and satisfy all requirements even in presence of exceptional situations. If an exception occurs, it will not throw the exception further up. (Best level of exception safety.)
2. **Commit or rollback semantics**, also known as **strong exception safety** or **no-change guarantee**: Operations can fail, but failed operations are guaranteed to have no side effects so all data retain original values.
3. **Basic exception safety**: Partial execution of failed operations can cause side effects, but invariants on the state are preserved. Any stored data will contain valid values even if data has different values now from before the exception.
4. **Minimal exception safety** also known as **no-leak guarantee**: Partial execution of failed operations may store invalid data but will not cause a crash, and no resources get leaked.
5. **No exception safety**: No guarantees are made. (Worst level of exception safety)

Partial handling

Consider the following case:

```
void g()
{
    throw "Exception";
}

void f()
{
    int* pI = new int(0);
    g();
    delete pI;
}

int main()
{
    f();
    return 0;
}
```

36 HTTP://EN.WIKIPEDIA.ORG/WIKI/MEMORY%20LEAK

37 HTTP://EN.WIKIPEDIA.ORG/WIKI/INVARIANT%20%28COMPUTER%20SCIENCE%29

Can you see the problem in this code? If `g()` throws an exception, the variable `pI` is never deleted and we have a memory leak.

To prevent the memory leak, `f()` must catch the exception, and delete `pI`. But `f()` can't handle the exception, it doesn't know how!

What is the solution then? `f()` shall catch the exception, and then re-throw it:

```
void g()
{
    throw "Exception";
}

void f()
{
    int* pI = new int(0)

    try
    {
        g();
    }
    catch (...)
    {
        delete pI;
        throw; // This empty throw re-throws the exception we caught
               // An empty throw can only exist in a catch block
    }

    delete pI;
}

int main()
{
    f();
    return 0;
}
```

There's a better way though; using **RAII** classes to avoid the need to use exception handling.

Guards

If you plan to use exceptions in your code, you must always try to write your code in an exception safe manner. Let's see some of the problems that can occur:

Consider the following code:

```
void g()
{
    throw std::exception();
}

void f()
```

```
{
    int* pI = new int(2);

    *pI = 3;
    g();
    // Oops, if an exception is thrown, pI is never deleted
    // and we have a memory leak
    delete pI;
}

int main()
{
    try
    {
        f();
    }
    catch(...)
    { }

    return 0;
}
```

Can you see the problem in this code? When an exception is thrown, we will never run the line that deletes `pI`!

What's the solution to this? Earlier we saw a solution based on `f()` ability to catch and re-throw. But there is a neater solution using the 'stack unwinding' mechanism. But 'stack unwinding' only applies to destructors for objects, so how can we use it?

We can write a simple wrapper class:

```
// Note: This type of class is best implemented using templates, discussed in
the next chapter.
class IntDeleter {
public:
   IntDeleter(int* piValue)
   {
       m_piValue = piValue;
   }

   ~IntDeleter()
   {
       delete m_piValue;
   }

   // operator *, enables us to dereference the object and use it
   // like a regular pointer.
   int&  operator *()
   {
       return *m_piValue;
   }

private:
```

```
    int* m_piValue;
};
```

The new version of `f()`:

```
void f()
{
  IntDeleter pI(new int(2));

  *pI = 3;
  g();
  // No need to delete pI, this will be done in destruction.
  // This code is also exception safe.
}
```

The pattern presented here is called a *guard*. A guard is very useful in other cases, and it can also help us make our code more exception safe. The guard pattern is similar to a *finally* block in other languages.

Note that the C++ Standard Library provides a templated guard by the name of auto_ptr.

Exception hierarchy

You may throw as exception an object (like a class or string), a pointer (like `char*`), or a primitive (like `int`). So, which should you choose? You should throw objects, as they ease the handling of exceptions for the programmer. It is common to create a class hierarchy of exception classes:

- class MyApplicationException {};
  - class MathematicalException : public MyApplicationException {};
    - class DivisionByZeroException : public MathematicalException {};
  - class InvalidArgumentException : public MyApplicationException {};

An example:

```
float divide(float fNumerator, float fDenominator)
{
    if (fDenominator == 0.0)
    {
        throw DivisionByZeroException();
    }

    return fNumerator/fDenominator;
}

enum MathOperators {DIVISION, PRODUCT};

float operate(int iAction, float fArgLeft, float fArgRight)
{
```

```
    if (iAction == DIVISION)
    {
        return divide(fArgLeft, fArgRight);
    }
    else if (iAction == PRODUCT))
    {
        // call the product function
        // ...
    }

    // No match for the action! iAction is an invalid agument
    throw InvalidArgumentException();
}

int main(int iArgc, char* a_pchArgv[])
{
    try
    {
        operate(atoi(a_pchArgv[0]), atof(a_pchArgv[1]), atof(a_pchArgv[2]));
    }
    catch(MathematicalException& )
    {
        // Handle Error
    }
    catch(MyApplicationException& )
    {
        // This will catch in InvalidArgumentException too.
        // Display help to the user, and explain about the arguments.
    }

    return 0;
}
```

Note:
The order of the catch blocks is important. A thrown object (say, InvalidArgumentException) can be caught in a catch block of one of its super-classes. (e.g. `catch (MyApplicationException& )` will catch it too). This is why it is important to place the catch blocks of derived classes before the catch block of their super classes.

5.5.5 Exception specifications

The range of exceptions that can be thrown by a function are an important part of that function's public interface. Without this information, you would have to assume that any exception could occur when calling any function, and consequently write code that was extremely defensive. Knowing the list of exceptions that can be thrown, you can simplify your code since it doesn't need to handle every case.

This exception information is specifically part of the *public* interface. Users of a class don't need to know anything about the way it is implemented, but they do need to know about the exceptions that can be thrown, just as they need to know the number and type of parameters to a member function. One way of providing this information to clients of a library is via code documentation, but this needs to be manually updated very carefully. Incorrect exception information is worse than none at all, since you may end up writing code that is less exception-safe than you intended to.

C++ provides another way of recording the exception interface, by means of *exception specifications*. An exception specification is parsed by the compiler, which provides a measure of automated checking. An exception specification can be applied to any function, and looks like this:

```
double divide(double dNumerator, double dDenominator) throws
 (DivideByZeroException);
```

You can specify that a function cannot throw any exceptions by using an empty exception specification:

```
void safeFunction(int iFoo) throws();
```

Shortcomings of exception specifications

C++ does not programmatically enforce exception specifications at compile time. For example, the following code is legal:

```
void DubiousFunction(int iFoo) throws()
{
    if (iFoo < 0)
    {
        throw RangeException();
    }
}
```

Rather than checking exception specifications at compile time, C++ checks them at run time, which means that you might not realize that you have an inaccurate exception specification until testing or, if you are unlucky, when the code is already in production.

If an exception is thrown at run time that propagates out of a function that doesn't allow the exception in its exception specification, the exception will not propagate any further and instead, the function `RangeException()` will be called. The

`RangeException()` function doesn't return, but can throw a different type of exception that may (or may not) satisfy the exception specification and allow exception handling to carry on normally. If this still doesn't recover the situation, the program will be terminated.

Many people regard the behavior of attempting to translate exceptions at run time to be worse than simply allowing the exception to propagate up the stack to a caller who may be able to handle it. The fact that the exception specification has been violated does not mean that the caller *can't* handle the situation, only that the author of the code didn't expect it. Often there will be a `catch (...)` block somewhere on the stack that can deal with *any* exception.

Note:
Some coding standards require that exception specifications are not used. In the upcoming C++ language standard (C++0x), the use of exception specifications as specified in the current version of the standard (C++03), is deprecated.

38

5.6 Run-Time Type Information (RTTI)

RTTI refers to the ability of the system to report on the dynamic type of an object and to provide information about that type at runtime (as opposed to at compile time), when utilized consistently can be a powerful tool to ease the work of the programmer in managing resources.

5.6.1 `dynamic_cast`

Consider what you have already learned about the `dynamic_cast` keyword and let's say that we have the following class hierarchy:

```
class Interface
{
public:
       virtual void GenericOp() = 0;// pure virtual function
};

class SpecificClass : public Interface
```

38 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

```
{
public:
        virtual void GenericOp();
        virtual void SpecificOp();
};
```

Let's say that we also have a pointer of type `Interface*`, like so:

```
Interface* ptr_interface;
```

Supposing that a situation emerges that we are forced to presume but have no guarantee that the pointer points to an object of type `SpecificClass` and we would like to call the member `SpecificOp()` of that class. To dynamically convert to a derived type we can use `dynamic_cast`, like so:

```
SpecificClass* ptr_specific = dynamic_cast<SpecificClass*>(ptr_interface);
if( ptr_specific ){
    // our suspicions are confirmed -- it really was a SpecificClass
    ptr_specific->SpecificOp();
}else{
    // our suspicions were incorrect -- it is definitely not a SpecificClass.
    // The ptr_interface points to an instance of some other child class of the
 base InterfaceClass.
};
ptr_interface->GenericOp();
```

With `dynamic_cast`, the program converts the base class pointer to a derived class pointer and allows the derived class members to be called. Be very careful, however: if the pointer that you are trying to cast is not of the correct type, then `dynamic_cast` will return a null pointer.

We can also use `dynamic_cast` with references.

```
SpecificClass& ref_specific = dynamic_cast<SpecificClass&>(ref_interface);
```

This works almost in the same way as pointers. However, if the real type of the object being cast is not correct then `dynamic_cast` will not return null (there's no such thing as a null reference). Instead, it will throw a `std::bad_cast` exception.

5.6.2 `typeid`

Syntax

```
    typeid( object );
```

The **typeid** operator, used to determine the class of an object at runtime. It returns a reference to a `std::type_info` object, which exists until the end of the program, that describes the "object". If the "object" is a dereferenced null pointer, then the operation will throw a `std::bad_typeid` exception.

Objects of class `std::bad_typeid` are derived from `std::exception`, and thrown by **typeid** and others.

Note:
The C++98 standard requires that header file <*typeinfo*> to be included before operator **typeid** is used within a compilation unit. Otherwise, the program is considered ill-formed.

The use of `typeid` is often preferred over `dynamic_cast<`*class_type*`>` in situations where just the class information is needed, because `typeid`, applied on a type or non de-referenced value is a CONSTANT-TIME[39] procedure, whereas `dynamic_cast` must traverse the class derivation lattice of its argument at runtime. Though one should never rely on the exact content, like for example returned by `std::type_info::name()`, as this is implementation specific with respect to the compile.

It is generally only useful to use **typeid** on the dereference of a pointer or reference (i.e. `typeid(*ptr)` or `typeid(ref)`) to an object of polymorphic class type (a class with at least one VIRTUAL MEMBER FUNCTION[40]). This is because these are the only expressions that are associated with run-time type information. The type of any other expression is statically known at compile time.

Example

```
#include <iostream>
#include <typeinfo>  //for 'typeid' to work

class Person {
public:
   // ... Person members ...
   virtual ~Person() {}
};

class Employee : public Person {
   // ... Employee members ...
};
```

39 HTTP://EN.WIKIPEDIA.ORG/WIKI/CONSTANT%20TIME
40 Chapter 4.3.1 on page 394

```
int main () {
   Person person;
   Employee employee;
   Person *ptr = &employee;
   // The string returned by typeid::name is implementation-defined
   std::cout << typeid(person).name() << std::endl;   // Person (statically known
 at compile-time)
   std::cout << typeid(employee).name() << std::endl; // Employee (statically
 known at compile-time)
   std::cout << typeid(ptr).name() << std::endl;      // Person * (statically
 known at compile-time)
   std::cout << typeid(*ptr).name() << std::endl;     // Employee (looked up
 dynamically at run-time
                                                     //           because it is
 the dereference of a
                                                     //           pointer to a
 polymorphic class)
}
```

Output (exact output varies by system):

```
Person
Employee
Person*
Employee
```

In RTTI it is used in this setup:

```
const std::type_info& info = typeid(object_expression);
```

Sometimes we need to know the exact type of an object. The `typeid` operator returns a reference to a standard class `std::type_info` that contains information about the type. This class provides some useful members including the == and `!=` operators. The most interesting method is probably:

```
const char* std::type_info::name() const;
```

This member function returns a pointer to a C-style string with the name of the object type. For example, using the classes from our earlier example:

```
const std::type_info &info = typeid(*ptr_interface);
std::cout << info.name() << std::endl;
```

This program would print something like[41] `SpecificClass` because that is the dynamic type of the pointer `ptr_interface`.

`typeid` is actually an operator rather than a function, as it can also act on types:

41 (The exact string returned by std::type_info::name() is compiler-dependent).

```
const std::type_info& info = typeid(type);
```

for example (and somewhat circularly)

```
const std::type_info& info = typeid(std::type_info);
```

will give a `type_info` object which describes `type_info` objects. This latter use is not RTTI, but rather CTTI (compile-time type identification).

5.6.3 Limitations

There are some limitations to RTTI. First, RTTI can only be used with *polymorphic types*. That means that your classes must have at least one virtual function, either directly or through inheritance. Second, because of the additional information required to store types some compilers require a special switch to enable RTTI.

Note that references to pointers will not work under RTTI:

```
void example( int*& refptrTest )
{
        std::cout << "What type is *&refptrTest : " << typeid( refptrTest
 ).name() << std::endl;
}
```

Will report `int*`, as `typeid()` does not support reference types.

5.6.4 Misuses of RTTI

RTTI should only be used sparingly in C++ programs. There are several reasons for this. Most importantly, other language mechanisms such as polymorphism and templates are almost always superior to RTTI. As with everything, there are exceptions, but the usual rule concerning RTTI is more or less the same as with `goto` statements. Do not use it as a shortcut around proper, more robust design. Only use RTTI if you have a very good reason to do so and only use it if you know what you are doing.

42

42 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

5.7 Chapter Summary

1. TEMPLATES[43]
   a) TEMPLATE META-PROGRAMMING (TMP)[44]
2. STANDARD TEMPLATE LIBRARY (STL)[45]
3. SMART POINTERS[46]
4. EXCEPTION HANDLING[47]
5. RUN-TIME TYPE INFORMATION (RTTI)[48]

4[49]

4[50]

43 Chapter 5 on page 483
44 Chapter 5.1.4 on page 492
45 Chapter 5.1.5 on page 499
46 Chapter 5.2.6 on page 515
47 Chapter 5.4 on page 517
48 Chapter 5.5.5 on page 530
49 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3A
50 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

6 Beyond the Standard

6.1 Resource Acquisition Is Initialization (RAII)

The RAII technique is often used for controlling thread locks in multi-threaded applications. Another typical example of RAII is file operations, e.g. the C++ standard library's file-streams. An input file stream is opened in the object's constructor, and it is closed upon destruction of the object. Since C++ allows objects to be allocated on the STACK[1], C++'s scoping mechanism can be used to control file access.

With RAII we can use, for instance, Class destructors to guarantee clean up, similar to the *finally* keyword in other languages. Doing this automates the task and so avoids errors but gives the freedom not to use it.

RAII is also used (as shown in the example below) to ensure exception safety. RAII makes it possible to avoid resource leaks without extensive use of `try`/`catch` blocks and is widely used in the software industry.

The ownership of dynamically allocated memory (memory allocated with `new`) can be controlled with RAII. For this purpose, the C++ Standard Library defines AUTO PTR[2]. Furthermore, lifetime of shared objects can be managed by a smart pointer with shared-ownership semantics such as `boost::shared_ptr` defined in C++ by the BOOST LIBRARY[3] or policy based `Loki::SmartPtr` from LOKI LIBRARY[4].

The following RAII class is a lightweight wrapper to the C standard library file system calls.

```
#include <cstdio>

// exceptions
class file_error { } ;
class open_error : public file_error { } ;
```

1 HTTP://EN.WIKIPEDIA.ORG/WIKI/CALL%20STACK
2 HTTP://EN.WIKIPEDIA.ORG/WIKI/AUTO%20PTR
3 Chapter 6.4.2 on page 588
4 HTTP://EN.WIKIPEDIA.ORG/WIKI/LOKI%20%28C%2B%2B%29

```
class close_error : public file_error { } ;
class write_error : public file_error { } ;

class file
{
public:
    file( const char* filename )
        :
        m_file_handle(std::fopen(filename, "w+"))
    {
        if( m_file_handle == NULL )
        {
            throw open_error() ;
        }
    }

    ~file()
    {
        std::fclose(m_file_handle) ;
    }

    void write( const char* str )
    {
        if( std::fputs(str, m_file_handle) == EOF )
        {
            throw write_error() ;
        }
    }

    void write( const char* buffer, std::size_t num_chars )
    {
        if( num_chars != 0
            &&
            std::fwrite(buffer, num_chars, 1, m_file_handle) == 0 )
        {
            throw write_error() ;
        }
    }

private:
    std::FILE* m_file_handle ;

    // copy and assignment not implemented; prevent their use by
    // declaring private.
    file( const file & ) ;
    file & operator=( const file & ) ;
} ;
```

This RAII class can be used as follows :

```
void example_with_RAII()
{
  // open file (acquire resource)
  file logfile("logfile.txt") ;

  logfile.write("hello logfile!") ;
  // continue writing to logfile.txt ...
```

```
  // logfile.txt will automatically be closed because logfile's
  // destructor is always called when example_with_RAII() returns or
  // throws an exception.
}
```

Without using RAII, each function using an output log would have to manage the file explicitly. For example, an equivalent implementation without using RAII is this:

```
void example_without_RAII()
{
  // open file
  std::FILE* file_handle = std::fopen("logfile.txt", "w+") ;

  if( file_handle == NULL )
  {
    throw open_error() ;
  }

  try
  {

    if( std::fputs("hello logfile!", file_handle) == EOF )
    {
      throw write_error() ;
    }

    // continue writing to logfile.txt ... do not return
    // prematurely, as cleanup happens at the end of this function
  }
  catch(...)
  {
    // manually close logfile.txt
    std::fclose(file_handle) ;

    // re-throw the exception we just caught
    throw ;
  }

  // manually close logfile.txt
  std::fclose(file_handle) ;
}
```

The implementation of `file` and `example_without_RAII()` becomes more complex if `fopen()` and `fclose()` could potentially throw exceptions; `example_with_RAII()` would be unaffected, however.

The essence of the RAII idiom is that the class `file` encapsulates the management of any finite resource, like the `FILE*` file handle. It guarantees that the resource will properly be disposed of at function exit. Furthermore, `file` instances guarantee that a valid log file is available (by throwing an exception if the file could not be opened).

There's also a big problem in the presence of exceptions: in `example_without_RAII()`, if more than one resource were allocated, but an exception was to be thrown between their allocations, there's no general way to know which resources need to be released in the final `catch` block - and releasing a not-allocated resource is usually a bad thing. RAII takes care of this problem; the automatic variables are destructed in the reverse order of their construction, and an object is only destructed if it was fully constructed (no exception was thrown inside its constructor). So `example_without_RAII()` can never be as safe as `example_with_RAII()` without special coding for each situation, such as checking for invalid default values or nesting try-catch blocks. Indeed, it should be noted that `example_without_RAII()` contained resource bugs in previous versions of this article.

This frees `example_with_RAII()` from explicitly managing the resource as would otherwise be required. When several functions use `file`, this simplifies and reduces overall code size and helps ensure program correctness.

`example_without_RAII()` resembles the idiom used for resource management in non-RAII languages such as Java. While Java's *try-finally* blocks allow for the correct release of resources, the burden nonetheless falls on the programmer to ensure correct behavior, as each and every function using `file` may explicitly demand the destruction of the log file with a *try-finally* block.

5

6.2 Garbage collection

Garbage collection is a form of automatic memory management. The garbage collector or collector attempts to reclaim garbage, or memory used by objects that will never be accessed or mutated again by the application.

Tracing garbage collectors require some implicit runtime overhead that may be beyond the control of the programmer, and can sometimes lead to performance problems. For example, commonly used Stop-The-World garbage collectors, which pause program execution at arbitrary times, may make garbage collecting languages inappropriate for some embedded systems, high-performance server software, and applications with real-time needs.

A more fundamental issue is that garbage collectors violate locality of reference, since they deliberately go out of their way to find bits of memory that haven't been

5 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

accessed recently. The performance of modern computer architectures is increasingly tied to caching, which depends on the assumption of locality of reference for its effectiveness. Some garbage collection methods result in better locality of reference than others. Generational garbage collection is relatively cache-friendly, and copying collectors automatically defragment memory helping to keep related data together. Nonetheless, poorly timed garbage collection cycles could have a severe performance impact on some computations, and for this reason many runtime systems provide mechanisms that allow the program to temporarily suspend, delay or activate garbage collection cycles.

Despite these issues, for many practical purposes, allocation/deallocation-intensive algorithms implemented in modern garbage collected languages can actually be faster than their equivalents using explicit memory management (at least without heroic optimizations by an expert programmer). A major reason for this is that the garbage collector allows the runtime system to amortize allocation and deallocation operations in a potentially advantageous fashion. For example, consider the following program in C++:

```
#include <iostream>

class A {
  int x;
public:
  A() { x = 0; ++x; }
};

int main() {
 for (int i = 0; i < 1000000000; ++i) {
   A *a = new A();
   delete a;
 }
 std::cout << "DING!" << std::endl;
}
```

One of more widely used libraries that provides this function is HANS BOEHM'S CONSERVATIVE GC[6]. As we have seen earlier C++ also supports a powerful idiom called *RAII (resource acquisition is initialization)*[7] that can be used to safely and automatically manage resources including memory.

8

6 HTTP://WWW.HPL.HP.COM/PERSONAL/HANS_BOEHM/GC/

7 Chapter 6 on page 537

8 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

6.3 Programming Patterns

"To understand is to perceive patterns"
—ISAIAH BERLIN[9]

Software design patterns are abstractions that help structure system designs. While not new, since the concept was already described by CHRISTOPHER ALEXANDER[10] in its architectural theories, it only gathered some traction in programming due to the publication of DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE[11] book in October 1994 by ERICH GAMMA[12], RICHARD HELM[13], RALPH JOHNSON[14] and JOHN VLISSIDES[15], known as the **Gang of Four (GoF)**, that identifies and describes 23 classic software design patterns.

A design pattern is neither a static solution, nor is it an algorithm. A **pattern** is a way to describe and address by name (mostly a simplistic description of its goal), a repeatable solution or approach to a common design problem, that is, a common way to solve a generic problem (how generic or complex, depends on how restricted the target goal is). Patterns can emerge on their own or by design. This is why design patterns are useful as an abstraction over the implementation and a help at design stage. With this concept, an easier way to facilitate communication over a design choice as normalization technique is given so that every person can share the design concept.

Depending on the design problem they address, design patterns can be classified in different categories, of which the main categories are:

- CREATIONAL PATTERNS[16]
- STRUCTURAL PATTERNS[17]
- BEHAVIORAL PATTERNS[18].

9 http://en.wikipedia.org/wiki/Isaiah%20Berlin
10 http://en.wikipedia.org/wiki/Christopher%20Alexander
11 http://en.wikipedia.org/wiki/Design%20Patterns
12 http://en.wikipedia.org/wiki/Erich%20Gamma
13 http://en.wikipedia.org/wiki/Richard%20Helm
14 http://en.wikipedia.org/wiki/Ralph%20Johnson
15 http://en.wikipedia.org/wiki/John%20Vlissides
16 Chapter 6.3 on page 543
17 Chapter 6.3.1 on page 559
18 Chapter 6.3.2 on page 564

Patterns are commonly found in objected-oriented programming languages like C++ or Java. They can be seen as a template for how to solve a problem that occurs in many different situations or applications. It is not code reuse, as it usually does not specify code, but code can be easily created from a design pattern. Object-oriented design patterns typically show relationships and interactions between classes or objects without specifying the final application classes or objects that are involved.

Each design pattern consists of the following parts:

Problem/requirement

To use a design pattern, we need to go through a mini analysis design that may be coded to test out the solution. This section states the requirements of the problem we want to solve. This is usually a common problem that will occur in more than one application.

Forces

This section states the technological boundaries, that helps and guides the creation of the solution.

Solution

This section describes how to write the code to solve the above problem. This is the design part of the design pattern. It may contain class diagrams, sequence diagrams, and or whatever is needed to describe how to code the solution.

Design patterns can be considered as a standardization of commonly agreed best practices to solve specific design problems. One should understand them as a way to implement good design patterns within applications. Doing so will reduce the use of inefficient and obscure solutions. Using design patterns speeds up your design and helps to communicate it to other programmers.

6.3.1 Creational Patterns

In SOFTWARE ENGINEERING[19], **creational design patterns** are DESIGN PATTERNS[20] that deal with OBJECT CREATION[21] mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result

19 HTTP://EN.WIKIPEDIA.ORG/WIKI/SOFTWARE%20ENGINEERING

20 HTTP://EN.WIKIPEDIA.ORG/WIKI/DESIGN%20PATTERN%20%28COMPUTER%20SCIENCE%29

21 HTTP://EN.WIKIPEDIA.ORG/WIKI/OBJECT%20LIFETIME

in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

In this section of the book we assume that the reader has enough familiarity with functions, global variables, stack vs. heap, classes, pointers, and static member functions as introduced before.

As we will see there are several creational design patterns, and all will deal with a specific implementation task, that will create a higher level of abstraction to the code base, we will now cover each one.

Builder

The Builder Creational Pattern is used to separate the construction of a complex object from its representation so that the same construction process can create different objects representations.

Problem

We want to construct a complex object, however we do not want to have a complex constructor member or one that would need many arguments.

Solution

Define an intermediate object whose member functions define the desired object part by part before the object is available to the client. Build Pattern lets us defer the construction of the object until all the options for creation have been specified.

```
#include <string>
#include <iostream>

using namespace std;

// "Product"
class Pizza
{
        public:
                void setDough(const string& dough)
                {
                        m_dough = dough;
                }
                void setSauce(const string& sauce)
                {
                        m_sauce = sauce;
                }
                void setTopping(const string& topping)
                {
                        m_topping = topping;
                }
```

```
                void open() const
                {
                        cout << "Pizza with " << m_dough << " dough, " << m_sauce << " sauce
 and "
                        << m_topping << " topping. Mmm." << endl;
                }
        private:
                string m_dough;
                string m_sauce;
                string m_topping;
};

// "Abstract Builder"
class PizzaBuilder
{
        public:
                Pizza* getPizza()
                {
                        return m_pizza;
                }
                void createNewPizzaProduct()
                {
                        m_pizza = new Pizza;
                }
                virtual void buildDough() = 0;
                virtual void buildSauce() = 0;
                virtual void buildTopping() = 0;
        protected:
                Pizza* m_pizza;
};

//------------------------------------------------------------------------

class HawaiianPizzaBuilder : public PizzaBuilder
{
        public:
                virtual void buildDough()
                {
                        m_pizza->setDough("cross");
                }
                virtual void buildSauce()
                {
                        m_pizza->setSauce("mild");
                }
                virtual void buildTopping()
                {
                        m_pizza->setTopping("ham+pineapple");
                }
};

class SpicyPizzaBuilder : public PizzaBuilder
{
        public:
                virtual void buildDough()
                {
                        m_pizza->setDough("pan baked");
                }
                virtual void buildSauce()
```

```
                {
                        m_pizza->setSauce("hot");
                }
                virtual void buildTopping()
                {
                        m_pizza->setTopping("pepperoni+salami");
                }
};

//-----------------------------------------------------------------------

class Cook
{
        public:
                void setPizzaBuilder(PizzaBuilder* pb)
                {
                        m_pizzaBuilder = pb;
                }
                Pizza* getPizza()
                {
                        return m_pizzaBuilder->getPizza();
                }
                void constructPizza()
                {
                        m_pizzaBuilder->createNewPizzaProduct();
                        m_pizzaBuilder->buildDough();
                        m_pizzaBuilder->buildSauce();
                        m_pizzaBuilder->buildTopping();
                }
        private:
                PizzaBuilder* m_pizzaBuilder;
};

int main()
{
        Cook cook;
        PizzaBuilder* hawaiianPizzaBuilder = new HawaiianPizzaBuilder;
        PizzaBuilder* spicyPizzaBuilder   = new SpicyPizzaBuilder;

        cook.setPizzaBuilder(hawaiianPizzaBuilder);
        cook.constructPizza();

        Pizza* hawaiian = cook.getPizza();
        hawaiian->open();

        cook.setPizzaBuilder(spicyPizzaBuilder);
        cook.constructPizza();

        Pizza* spicy = cook.getPizza();
        spicy->open();

        delete hawaiianPizzaBuilder;
        delete spicyPizzaBuilder;
        delete hawaiian;
        delete spicy;
}
```

Factory

Definition: A utility class that creates an instance of a class from a family of derived classes

Abstract Factory

Definition: A utility class that creates an instance of several families of classes. It can also return a factory for a certain group.

Factory Method

The Factory Design Pattern is useful in a situation that requires the creation of many different types of objects, all derived from a common base type. The Factory Method defines a method for creating the objects, which subclasses can then override to specify the derived type that will be created. Thus, at run time, the Factory Method can be passed a description of a desired object (e.g., a string read from user input) and return a base class pointer to a new instance of that object. The pattern works best when a well-designed interface is used for the base class, so there is no need to cast the returned object.

Problem

We want to decide at run time what object is to be created based on some configuration or application parameter. When we write the code, we do not know what class should be instantiated.

Solution

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

In the following example, a factory method is used to create laptop or desktop computer objects at run time.

Let's start by defining `Computer`, which is an abstract base class (interface) and its derived classes: `Laptop` and `Desktop`.

```
class Computer
{
public:
    virtual void Run() = 0;
    virtual void Stop() = 0;
};
class Laptop: public Computer
```

```
{
public:
    virtual void Run(){mHibernating = false;}
    virtual void Stop(){mHibernating = true;}
private:
    bool mHibernating; // Whether or not the machine is hibernating
};
class Desktop: public Computer
{
public:
    virtual void Run(){mOn = true;}
    virtual void Stop(){mOn = false;}
private:
    bool mOn; // Whether or not the machine has been turned on
};
```

The actual `ComputerFactory` class returns a `Computer`, given a real world description of the object.

```
class ComputerFactory
{
public:
    static Computer *NewComputer(const std::string &description)
    {
        if(description == "laptop")
            return new Laptop;
        if(description == "desktop")
            return new Desktop;
        return NULL;
    }
};
```

Let's analyze the benefits of this design. First, there is a compilation benefit. If we move the interface `Computer` into a separate header file with the factory, we can then move the implementation of the `NewComputer()` function into a separate implementation file. Now the implementation file for `NewComputer()` is the only one that requires knowledge of the derived classes. Thus, if a change is made to any derived class of `Computer`, or a new `Computer` subtype is added, the implementation file for `NewComputer()` is the only file that needs to be recompiled. Everyone who uses the factory will only care about the interface, which should remain consistent throughout the life of the application.

Also, if there is a need to add a class, and the user is requesting objects through a user interface, no code calling the factory may be required to change to support the additional computer type. The code using the factory would simply pass on the new string to the factory, and allow the factory to handle the new types entirely.

Imagine programming a video game, where you would like to add new types of enemies in the future, each of which has different AI functions and can update differently. By using a factory method, the controller of the program can call to

the factory to create the enemies, without any dependency or knowledge of the actual types of enemies. Now, future developers can create new enemies, with new AI controls and new drawing member functions, add it to the factory, and create a level which calls the factory, asking for the enemies by name. Combine this method with an XML[22] description of levels, and developers could create new levels without having to recompile their program. All this, thanks to the separation of creation of objects from the usage of objects.

Another example:

```
#include <stdexcept>
#include <iostream>
#include <memory>

class Pizza {
public:
    virtual int getPrice() const = 0;
};

class HamAndMushroomPizza : public Pizza {
public:
    virtual int getPrice() const { return 850; }
};

class DeluxePizza : public Pizza {
public:
    virtual int getPrice() const { return 1050; }
};

class HawaiianPizza : public Pizza {
public:
    virtual int getPrice() const { return 1150; }
};

class PizzaFactory {
public:
    enum PizzaType {
         HamMushroom,
         Deluxe,
         Hawaiian
    };

    static Pizza* createPizza(PizzaType pizzaType) {
        switch (pizzaType) {
            case HamMushroom:
                return new HamAndMushroomPizza();
            case Deluxe:
                return new DeluxePizza();
            case Hawaiian:
                return new HawaiianPizza();
        }
        throw "invalid pizza type.";
```

22 HTTP://EN.WIKIBOOKS.ORG/WIKI/XML

```
    }
};

/*
 * Create all available pizzas and print their prices
 */
void pizza_information( PizzaFactory::PizzaType pizzatype )
{
        Pizza* pizza = PizzaFactory::createPizza(pizzatype);
        std::cout << "Price of " << pizzatype << " is " << pizza->getPrice() <<
 std::endl;
        delete pizza;
}

int main ()
{
        pizza_information( PizzaFactory::HamMushroom );
        pizza_information( PizzaFactory::Deluxe );
        pizza_information( PizzaFactory::Hawaiian );
}
```

Prototype

A prototype pattern is used in software development when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects. This pattern is used, for example, when the inherent cost of creating a new object in the standard way (e.g., using the `new` keyword) is prohibitively expensive for a given application.

Implementation: Declare an abstract base class that specifies a pure virtual `clone()` method. Any class that needs a "polymorphic constructor" capability derives itself from the abstract base class, and implements the `clone()` operation.

Here the client code first invokes the factory method. This factory method, depending on the parameter, finds out concrete class. On this concrete class call to the `clone()` method is called and the object is returned by the factory method.

- This is sample code which is a sample implementation of Prototype method. We have the detailed description of all the components here.
  - `Record` class, which is a pure virtual class that has a pure virtual method `clone()`.
  - `CarRecord`, `BikeRecord` and `PersonRecord` as concrete implementation of a `Record` class.
  - An `enum` RECORD_TYPE_en as one to one mapping of each concrete implementation of `Record` class.

- `RecordFactory` class that has a `Factory` method `CreateRecord(...)`. This method requires an `enum` RECORD_TYPE_en as parameter and depending on this parameter it returns the concrete implementation of `Record` class.

```
/**
 * Implementation of Prototype Method
 **/
#include <iostream>
#include <map.h>
#include <string>

using namespace std;

enum RECORD_TYPE_en
{
  CAR,
  BIKE,
  PERSON
};

/**
 * Record is the Prototype
 */

class Record
{
  public :

    Record() {}

    virtual ~Record() {}

    virtual Record* clone()=0;

    virtual void print()=0;
};

/**
 * CarRecord is a Concrete Prototype
 */

class CarRecord : public Record
{
  private:
    string m_carName;
    int m_ID;

  public:
    CarRecord(string carName, int ID)
      : Record(), m_carName(carName),
        m_ID(ID)
    {
    }

    CarRecord(CarRecord& carRecord)
      : Record()
```

```
    {
      m_carName = carRecord.m_carName;
      m_ID = carRecord.m_ID;
    }

    ~CarRecord() {}

    Record* clone()
    {
      return new CarRecord(*this);
    }

    void print()
    {
      cout << "Car Record" << endl
        << "Name  : " << m_carName << endl
        << "Number: " << m_ID << endl << endl;
    }
};


/**
 * BikeRecord is the Concrete Prototype
 */

class BikeRecord : public Record
{
  private :
    string m_bikeName;

    int m_ID;

  public :
    BikeRecord(string bikeName, int ID)
      : Record(), m_bikeName(bikeName),
        m_ID(ID)
    {
    }

    BikeRecord(BikeRecord& bikeRecord)
      : Record()
    {
      m_bikeName = bikeRecord.m_bikeName;
      m_ID = bikeRecord.m_ID;
    }

    ~BikeRecord() {}

    Record* clone()
    {
      return new BikeRecord(*this);
    }

    void print()
    {
      cout << "Bike Record" << endl
        << "Name  : " << m_bikeName << endl
        << "Number: " << m_ID << endl << endl;
```

```
    }
};


/**
 * PersonRecord is the Concrete Prototype
 */

class PersonRecord : public Record
{
  private :
    string m_personName;

    int m_age;

  public :
    PersonRecord(string personName, int age)
      : Record(), m_personName(personName),
        m_age(age)
    {
    }

    PersonRecord(PersonRecord& personRecord)
      : Record()
    {
      m_personName = personRecord.m_personName;
      m_age = personRecord.m_age;
    }

    ~PersonRecord() {}

    Record* clone()
    {
      return new PersonRecord(*this);
    }

  void print()
  {
    cout << "Person Record" << endl
      << "Name : " << m_personName << endl
      << "Age  : " << m_age << endl << endl ;
  }
};


/**
 * RecordFactory is the client
 */

class RecordFactory
{
  private :
    map<RECORD_TYPE_en, Record* > m_recordReference;

  public :
    RecordFactory()
    {
      m_recordReference[CAR]  = new CarRecord("Ferrari", 5050);
```

```
      m_recordReference[BIKE] = new BikeRecord("Yamaha", 2525);
      m_recordReference[PERSON] = new PersonRecord("Tom", 25);
    }

    ~RecordFactory()
    {
      delete m_recordReference[CAR];
      delete m_recordReference[BIKE];
      delete m_recordReference[PERSON];
    }

    Record* createRecord(RECORD_TYPE_en enType)
    {
      return m_recordReference[enType]->clone();
    }
};

int main()
{
  RecordFactory* poRecordFactory = new RecordFactory();

  Record* poRecord;
  poRecord = poRecordFactory->createRecord(CAR);
  poRecord->print();
  delete poRecord;

  poRecord = poRecordFactory->createRecord(BIKE);
  poRecord->print();
  delete poRecord;

  poRecord = poRecordFactory->createRecord(PERSON);
  poRecord->print();
  delete poRecord;

  delete poRecordFactory;
  return 0;
}
```

Another example:

To implement the pattern, declare an abstract base class that specifies a pure virtual `clone()` member function. Any class that needs a "polymorphic constructor" capability derives itself from the abstract base class, and implements the `clone()` operation.

The client, instead of writing code that invokes the `new` operator on a hard-wired class name, calls the `clone()` member function on the prototype, calls a factory member function with a parameter designating the particular concrete derived class desired, or invokes the `clone()` member function through some mechanism provided by another design pattern.

```
class CPrototypeMonster
{
protected:
```

```
    CString           _name;
public:
    CPrototypeMonster();
    CPrototypeMonster( const CPrototypeMonster& copy );
    virtual ~CPrototypeMonster();

    virtual CPrototypeMonster*    Clone() const=0; // This forces every derived
class to provide an overload for this function.
    void        Name( CString name );
    CString    Name() const;
};

class CGreenMonster : public CPrototypeMonster
{
protected:
    int         _numberOfArms;
    double      _slimeAvailable;
public:
    CGreenMonster();
    CGreenMonster( const CGreenMonster& copy );
    ~CGreenMonster();

    virtual CPrototypeMonster*    Clone() const;
    void  NumberOfArms( int numberOfArms );
    void  SlimeAvailable( double slimeAvailable );

    int         NumberOfArms() const;
    double      SlimeAvailable() const;
};

class CPurpleMonster : public CPrototypeMonster
{
protected:
    int         _intensityOfBadBreath;
    double      _lengthOfWhiplikeAntenna;
public:
    CPurpleMonster();
    CPurpleMonster( const CPurpleMonster& copy );
    ~CPurpleMonster();

    virtual CPrototypeMonster*    Clone() const;

    void  IntensityOfBadBreath( int intensityOfBadBreath );
    void  LengthOfWhiplikeAntenna( double lengthOfWhiplikeAntenna );

    int       IntensityOfBadBreath() const;
    double    LengthOfWhiplikeAntenna() const;
};

class CBellyMonster : public CPrototypeMonster
{
protected:
    double      _roomAvailableInBelly;
public:
    CBellyMonster();
    CBellyMonster( const CBellyMonster& copy );
    ~CBellyMonster();
```

```
    virtual CPrototypeMonster*    Clone() const;

    void       RoomAvailableInBelly( double roomAvailableInBelly );
    double     RoomAvailableInBelly() const;
};

CPrototypeMonster* CGreenMonster::Clone() const
{
    return new CGreenMonster(*this);
}

CPrototypeMonster* CPurpleMonster::Clone() const
{
    return new CPurpleMonster(*this);
}

CPrototypeMonster* CBellyMonster::Clone() const
{
    return new CBellyMonster(*this);
}
```

A client of one of the concrete monster classes only needs a reference (pointer) to a `CPrototypeMonster` class object to be able to call the 'Clone' function and create copies of that object. The function below demonstrates this concept:

```
void DoSomeStuffWithAMonster( const CPrototypeMonster* originalMonster )
{
    CPrototypeMonster* newMonster = originalMonster->Clone();
    ASSERT( newMonster );

    newMonster->Name("MyOwnMonster");
    // Add code doing all sorts of cool stuff with the monster.
    delete newMonster;
}
```

Now originalMonster can be passed as a pointer to CGreenMonster, CPurpleMonster or CBellyMonster.

Singleton

The term **Singleton** refers to an object that can only be instantiated once. This pattern is generally used where a global variable would have otherwise been used. The main advantage of the singleton is that its existence is guaranteed. Other advantages of the design pattern include the clarity, from the unique access, that the object used is not on the local stack. Some of the downfalls of the object include that, like a global variable, it can be hard to tell what chunk of code corrupted memory, when a bug is found, since everyone has access to it.

Let's take a look at how a Singleton differs from other variable types.

Like a global variable, the Singleton exists outside of the scope of any functions. Traditional implementation uses a static member function of the Singleton class, which will create a single instance of the Singleton class on the first call, and forever return that instance. The following code example illustrates the elements of a C++ singleton class, that simply stores a single string.

```
class StringSingleton
{
public:
    // Some accessor functions for the class, itself
    std::string GetString() const
    {return mString;}
    void SetString(const std::string &newStr)
    {mString = newStr;}

    // The magic function, which allows access to the class from anywhere
    // To get the value of the instance of the class, call:
    //     StringSingleton::Instance().GetString();
    static StringSingleton &Instance()
    {
        // This line only runs once, thus creating the only instance in
existence
        static StringSingleton *instance = new StringSingleton;
        // dereferencing the variable here, saves the caller from having to use
        // the arrow operator, and removes tempation to try and delete the
        // returned instance.
        return *instance; // always returns the same instance
    }

private:
    // We need to make some given functions private to finish the definition of
the singleton
    StringSingleton(){} // default constructor available only to members or
friends of this class

    // Note that the next two functions are not given bodies, thus any attempt
    // to call them implicitly will return as compiler errors. This prevents
    // accidental copying of the only instance of the class.
    StringSingleton(const StringSingleton &old); // disallow copy constructor
    const StringSingleton &operator=(const StringSingleton &old); //disallow
assignment operator

    // Note that although this should be allowed,
    // some compilers may not implement private destructors
    // This prevents others from deleting our one single instance, which was
otherwise created on the heap
    ~StringSingleton(){}
private: // private data for an instance of this class
    std::string mString;
};
```

Variations of Singletons:

Applications of Singleton Class:

One common use of the singleton design pattern is for application configurations. Configurations may need to be accessible globally, and future expansions to the application configurations may be needed. The subset C's closest alternative would be to create a single global `struct`. This had the lack of clarity as to where this object was instantiated, as well as not guaranteeing the existence of the object.

Take, for example, the situation of another developer using your singleton inside the constructor of their object. Then, yet another developer decides to create an instance of the second class in the global scope. If you had simply used a global variable, the order of linking would then matter. Since your global will be accessed, possibly before main begins executing, there is no definition as to whether the global is initialized, or the constructor of the second class is called first. This behavior can then change with slight modifications to other areas of code, which would change order of global code execution. Such an error can be very hard to debug. But, with use of the singleton, the first time the object is accessed, the object will also be created. You now have an object which will always exist, in relation to being used, and will never exist if never used.

A second common use of this class is in updating old code to work in a new architecture. Since developers may have used globals liberally, moving them into a single class and making it a singleton, can be an intermediary step to bring the program inline to stronger object oriented structure.

Another example:

```
#include <iostream>
using namespace std;

/* Place holder for thread synchronization mutex */
class Mutex
{   /* placeholder for code to create, use, and free a mutex */
};

/* Place holder for thread synchronization lock */
class Lock
{   public:
        Lock(Mutex& m) : mutex(m) { /* placeholder code to acquire the mutex */ }
        ~Lock() { /* placeholder code to release the mutex */ }
    private:
        Mutex & mutex;
};

class Singleton
{   public:
        static Singleton* GetInstance();
        int a;
        ~Singleton() { cout << "In Dtor" << endl; }

    private:
        Singleton(int _a) : a(_a) { cout << "In Ctor" << endl; }
```

```
        static Mutex mutex;

        // Not defined, to prevent copying
        Singleton(const Singleton& );
        Singleton& operator =(const Singleton& other);
};

Mutex Singleton::mutex;

Singleton* Singleton::GetInstance()
{
    Lock lock(mutex);

    cout << "Get Inst" << endl;

    // Initialized during first access
    static Singleton inst(1);

    return &inst;
}

int main()
{
    Singleton* singleton = Singleton::GetInstance();
    cout << "The value of the singleton: " << singleton->a << endl;
    return 0;
}
```

Note:
In the above example, the first call to `Singleton::GetInstance` will initialize the singleton instance. This example is for illustrative purposes only; for anything but a trivial example program, this code contains errors.

23

6.3.2 Structural Patterns

Adapter

Convert the interface of a class into another interface clients expect. **Adapter** lets classes work together that couldn't otherwise because of incompatible interfaces.

23 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

Bridge

The Bridge Pattern is used to separate out the interface from its implementation. Doing this gives the flexibility so that both can vary independently.

The following example will output:

```
API1.circle at 1:2 7.5
API2.circle at 5:7 27.5
```

```
#include <iostream>

using namespace std;

/* Implementor*/
class DrawingAPI {
  public:
   virtual void drawCircle(double x, double y, double radius) = 0;
   virtual ~DrawingAPI() {}
};

/* Concrete ImplementorA*/
class DrawingAPI1 : public DrawingAPI {
  public:
   void drawCircle(double x, double y, double radius) {
      cout << "API1.circle at " << x << ':' << y << ' ' << radius << endl;
   }
};

/* Concrete ImplementorB*/
class DrawingAPI2 : public DrawingAPI {
public:
   void drawCircle(double x, double y, double radius) {
      cout << "API2.circle at " << x << ':' << y << ' ' <<  radius << endl;
   }
};

/* Abstraction*/
class Shape {
  public:
   virtual ~Shape() {}
   virtual void draw() = 0;
   virtual void resizeByPercentage(double pct) = 0;
};

/* Refined Abstraction*/
class CircleShape : public Shape {
  public:
   CircleShape(double x, double y,double radius, DrawingAPI *drawingAPI) :
           m_x(x), m_y(y), m_radius(radius), m_drawingAPI(drawingAPI)
   {}
   void draw() {
      m_drawingAPI->drawCircle(m_x, m_y, m_radius);
   }
```

```
   void resizeByPercentage(double pct) {
      m_radius *= pct;
   }
  private:
   double m_x, m_y, m_radius;
   DrawingAPI *m_drawingAPI;
};

int main(void) {
   CircleShape circle1(1,2,3,new DrawingAPI1());
   CircleShape circle2(5,7,11,new DrawingAPI2());
   circle1.resizeByPercentage(2.5);
   circle2.resizeByPercentage(2.5);
   circle1.draw();
   circle2.draw();
   return 0;
}
```

Composite

Composite lets clients treat individual objects and compositions of objects uniformly. The Composite pattern can represent both the conditions. In this pattern, one can develop tree structures for representing part-whole hierarchies.

```
#include <vector>
#include <iostream> // std::cout
#include <memory> // std::auto_ptr
#include <algorithm> // std::for_each
#include <functional> // std::mem_fun
using namespace std;

class Graphic
{
public:
  virtual void print() const = 0;
  virtual ~Graphic() {}
};

class Ellipse : public Graphic
{
public:
  void print() const {
    cout << "Ellipse \n";
  }
};

class CompositeGraphic : public Graphic
{
public:
  void print() const {
    // for each element in graphicList_, call the print member function
    for_each(graphicList_.begin(), graphicList_.end(), mem_fun(&Graphic::print));
  }
```

```
  void add(Graphic *aGraphic) {
    graphicList_.push_back(aGraphic);
  }

private:
  vector<Graphic*>  graphicList_;
};

int main()
{
  // Initialize four ellipses
  const auto_ptr<Ellipse> ellipse1(new Ellipse());
  const auto_ptr<Ellipse> ellipse2(new Ellipse());
  const auto_ptr<Ellipse> ellipse3(new Ellipse());
  const auto_ptr<Ellipse> ellipse4(new Ellipse());

  // Initialize three composite graphics
  const auto_ptr<CompositeGraphic> graphic(new CompositeGraphic());
  const auto_ptr<CompositeGraphic> graphic1(new CompositeGraphic());
  const auto_ptr<CompositeGraphic> graphic2(new CompositeGraphic());

  // Composes the graphics
  graphic1->add(ellipse1.get());
  graphic1->add(ellipse2.get());
  graphic1->add(ellipse3.get());

  graphic2->add(ellipse4.get());

  graphic->add(graphic1.get());
  graphic->add(graphic2.get());

  // Prints the complete graphic (four times the string "Ellipse")
  graphic->print();
  return 0;
}
```

Decorator

The decorator pattern helps to attach additional behavior or responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. This is also called “Wrapper”.

Facade

The Facade Pattern hides the complexities of the system by providing an interface to the client from where the client can access the system on an unified interface. Facade defines a higher-level interface that makes the subsystem easier to use. For instance making one class method perform a complex process by calling several other classes.

Flyweight

It is the use of sharing mechanism by which you can avoid creating a large number of object instances to represent the entire system by using a smaller set fine-grained objects efficiently. A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight will act as an independent object in each context, becoming indistinguishable from an instance of the object that's not shared. To decide if some part of a program is a candidate for using Flyweights, consider whether it is possible to remove some data from the class and make it extrinsic.

Proxy

The Proxy Pattern will provide an object a surrogate or placeholder for another object to control access to it. It is used when you need to represent a complex object with a simpler one. If creation of an object is expensive, it can be postponed until the very need arises and meanwhile a simpler object can serve as a placeholder. This placeholder object is called the "Proxy" for the complex object.

Curiously Recurring Template

This technique is known more widely as a mixin. Mixins are described in the literature to be a powerful tool for expressing abstractions.

Interface-based Programming (IBP)

Interface-based programming is closely related with Modular Programming and Object-Oriented Programming, it defines the application as a collection of inter-coupled modules (interconnected and which plug into each other via interface). Modules can be unplugged, replaced, or upgraded, without the need of compromising the contents of other modules.

The total system complexity is greatly reduced. Interface Based Programming adds more to modular Programming in that it insists that Interfaces are to be added to these modules. The entire system is thus viewed as Components and the interfaces that helps them to co-act.

Interface-based Programming increases the *modularity* of the application and hence its maintainability at a later development cycles, especially when each module must be developed by different teams. It is a well-known methodology that has

been around for a long time and it is a core technology behind frameworks such as CORBA.

This is particularly convenient when third parties develop additional components for the established system. They just have to develop components that satisfy the interface specified by the parent application vendor.

Thus the publisher of the interfaces assures that he will not change the interface and the subscriber agrees to implement the interface as whole without any deviation. An interface is therefore said to be a *Contractual agreement* and the PROGRAMMING PARADIGM[24] based on this is termed as "interface based programming".

[25]

6.3.3 Behavioral Patterns

Chain of Responsibility

Chain of Responsibility pattern has the intent to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chains the receiving objects and passes the requests along the chain until an object handles it.

Command

Command pattern is an Object behavioral pattern that decouples sender and receiver by encapsulating a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undo-able operations. It can also be thought as an object oriented equivalent of call back method.

Call Back: It is a function that is registered to be called at later point of time based on user actions.

```
#include <iostream>

using namespace std;

/*the Command interface*/
class Command
{
```

24 HTTP://EN.WIKIPEDIA.ORG/WIKI/PROGRAMMING%20PARADIGM
25 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

```
public:
        virtual void execute()=0;
};

/*Receiver class*/
class Light {

public:
        Light() {  }

        void turnOn()
        {
                cout << "The light is on" << endl;
        }

        void turnOff()
        {
                cout << "The light is off" << endl;
        }
};

/*the Command for turning on the light*/
class FlipUpCommand: public Command
{
public:

        FlipUpCommand(Light& light):theLight(light)
        {

        }

        virtual void execute()
        {
                theLight.turnOn();
        }

private:
        Light& theLight;
};

/*the Command for turning off the light*/
class FlipDownCommand: public Command
{
public:
        FlipDownCommand(Light& light) :theLight(light)
        {

        }
        virtual void execute()
        {
                theLight.turnOff();
        }
private:
        Light& theLight;
};

class Switch {
public:
```

```
        Switch(Command& flipUpCmd, Command& flipDownCmd)
        :flipUpCommand(flipUpCmd),flipDownCommand(flipDownCmd)
        {

        }

        void flipUp()
        {
                flipUpCommand.execute();
        }

        void flipDown()
        {
                flipDownCommand.execute();
        }

private:
        Command& flipUpCommand;
        Command& flipDownCommand;
};


/*The test class or client*/
int main()
{
        Light lamp;
        FlipUpCommand switchUp(lamp);
        FlipDownCommand switchDown(lamp);

        Switch s(switchUp, switchDown);
        s.flipUp();
        s.flipDown();
}
```

Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator

The 'iterator' design pattern is used liberally within the STL for traversal of various containers. The full understanding of this will liberate a developer to create highly reusable and easily understandable data containers.

The basic idea of the iterator is that it permits the traversal of a container (like a pointer moving across an array). However, to get to the next element of a container, you need not know anything about how the container is constructed. This is the iterators job. By simply using the member functions provided by the iterator, you

can move, in the intended order of the container, from the first element to the last element.

Let us start by considering a traditional single dimensional array with a pointer moving from the start to the end. This example assumes knowledge of pointer arithmetic. Note that the use of "it" or "itr," henceforth, is a short version of "iterator."

```
const int ARRAY_LEN = 42;
int *myArray = new int[ARRAY_LEN];
// Set the iterator to point to the first memory location of the array
int *arrayItr = myArray;
// Move through each element of the array, setting it equal to its position in
the array
for(int i = 0; i < ARRAY_LEN; ++i)
{
   // set the value of the current location in the array
   *arrayItr = i;
   // by incrementing the pointer, we move it to the next position in the array.
   // This is easy for a contiguous memory container, since pointer arithmetic
   // handles the traversal.
   ++arrayItr;
}
// Do not be messy, clean up after yourself
delete[] myArray;
```

This code works very quickly for arrays, but how would we traverse a linked list, when the memory is not contiguous? Consider the implementation of a rudimentary linked list as follows:

```
class IteratorCannotMoveToNext{}; // Error class
class MyIntLList
{
public:
    // The Node class represents a single element in the linked list.
    // The node has a next node and a previous node, so that the user
    // may move from one position to the next, or step back a single
    // position. Notice that the traversal of a linked list is O(N),
    // as is searching, since the list is not ordered.
    class Node
    {
    public:
        Node():mNextNode(0),mPrevNode(0),mValue(0){}
        Node *mNextNode;
        Node *mPrevNode;
        int mValue;
    };
    MyIntLList():mSize(0)
    {}
    ~MyIntLList()
    {
        while(!Empty())
            pop_front();
```

```
    } // See expansion for further implementation;
    int Size() const {return mSize;}
    // Add this value to the end of the list
    void push_back(int value)
    {
        Node *newNode = new Node;
        newNode->mValue = value;
        newNode->mPrevNode = mTail;
        mTail->mNextNode = newNode;
        mTail = newNode;
        ++mSize;
    }
    // Remove the value from the beginning of the list
    void pop_front()
    {
        if(Empty())
            return;
        Node *tmpnode = mHead;
        mHead = mHead->mNextNode
        delete tmpnode;
        --mSize;
    }
    bool Empty()
    {return mSize == 0;}

    // This is where the iterator definition will go,
    // but lets finish the definition of the list, first

private:
    Node *mHead;
    Node *mTail;
    int mSize;
};
```

This linked list has non-contiguous memory, and is therefore not a candidate for pointer arithmetic. And we do not want to expose the internals of the list to other developers, forcing them to learn them, and keeping us from changing it.

This is where the iterator comes in. The common interface makes learning the usage of the container easier, and hides the traversal logic from other developers.

Let us examine the code for the iterator, itself.

```
    /*
      *  The iterator class knows the internals of the linked list, so that it
      *  may move from one element to the next. In this implementation, I have
      *  chosen the classic traversal method of overloading the increment
      *  operators. More thorough implementations of a bi-directional linked
      *  list would include decrement operators so that the iterator may move
      *  in the opposite direction.
      */
    class Iterator
    {
    public:
        Iterator(Node *position):mCurrNode(position){}
```

```
        // Prefix increment
        const Iterator &operator++()
        {
            if(mCurrNode == 0 || mCurrNode->mNextNode == 0)
                throw IteratorCannotMoveToNext();e
            mCurrNode = mCurrNode->mNextNode;
            return *this;
        }
        // Postfix increment
        Iterator operator++(int)
        {
            Iterator tempItr = *this;
            ++(*this);
            return tempItr;
        }
        // Dereferencing operator returns the current node, which should then
        // be dereferenced for the int. TODO: Check syntax for overloading
        // dereferencing operator
        Node * operator*()
        {return mCurrNode;}
        // TODO: implement arrow operator and clean up example usage following
    private:
        Node *mCurrNode;
    };
    // The following two functions make it possible to create
    // iterators for an instance of this class.
    // First position for iterators should be the first element in the
container.
    Iterator Begin(){return Iterator(mHead);}
    // Final position for iterators should be one past the last element in the
container.
    Iterator End(){return Iterator(0);}
```

With this implementation, it is now possible, without knowledge of the size of the container or how its data is organized, to move through each element in order, manipulating or simply accessing the data. This is done through the accessors in the MyIntLList class, Begin() and End().

```
// Create a list
MyIntLList mylist;
// Add some items to the list
for(int i = 0; i < 10; ++i)
    myList.push_back(i);
// Move through the list, adding 42 to each item.
for(MyIntLList::Iterator it = myList.Begin(); it != myList.End(); ++it)
    (*it)->mValue += 42;
```

The following program gives the implementation of iterator design pattern with a generic template:

```
/*************************************************************************/
/* Iterator.h                                                            */
/*************************************************************************/
#ifndef MY_ITERATOR_HEADER
#define MY_ITERATOR_HEADER
```

```
#include <iterator>
#include <vector>
#include <set>

////////////////////////////////////////////////////////////////////////
template<class T, class U>
class Iterator
{
public:
	typedef typename std::vector<T>::iterator iter_type;
	Iterator(U *pData):m_pData(pData){
		m_it = m_pData->m_data.begin();
	}

	void first()
	{
		m_it = m_pData->m_data.begin();
	}

	void next()
	{
		m_it++;
	}

	bool isDone()
	{
		return (m_it == m_pData->m_data.end());
	}

	iter_type current()
	{
		return m_it;
	}
private:
	U *m_pData;
	iter_type m_it;
};

template<class T, class U, class A>
class setIterator
{
public:
	typedef typename std::set<T,U>::iterator iter_type;

	setIterator(A *pData):m_pData(pData)
	{
		m_it = m_pData->m_data.begin();
	}

	void first()
	{
		m_it = m_pData->m_data.begin();
	}

	void next()
	{
		m_it++;
```

```
        }

        bool isDone()
        {
                return (m_it == m_pData->m_data.end());
        }

        iter_type current()
        {
                return m_it;
        }

private:
        A                       *m_pData;
        iter_type               m_it;
};
#endif


/*************************************************************************/
/* Aggregate.h                                                           */
/*************************************************************************/
#ifndef MY_DATACOLLECTION_HEADER
#define MY_DATACOLLECTION_HEADER
#include "Iterator.h"

template <class T>
class aggregate
{
        friend class Iterator<T, aggregate>;
public:
        void add(T a)
        {
                m_data.push_back(a);
        }

        Iterator<T, aggregate> *create_iterator()
        {
                return new Iterator<T, aggregate>(this);
        }


private:
        std::vector<T> m_data;
};
template <class T, class U>
class aggregateSet
{
        friend class setIterator<T, U, aggregateSet>;
public:
        void add(T a)
        {
                m_data.insert(a);
        }

        setIterator<T, U, aggregateSet> *create_iterator()
        {
                return new setIterator<T,U,aggregateSet>(this);
```

```
        }

        void Print()
        {
                copy(m_data.begin(), m_data.end(), std::ostream_iterator<T>(std::cout,
 "\n"));
        }

private:
        std::set<T,U> m_data;
};

#endif


/*************************************************************************/
/* Iterator Test.cpp                                                     */
/*************************************************************************/
#include <iostream>
#include <string>
#include "Aggregate.h"
using namespace std;

class Money
{
public:
        Money(int a = 0): m_data(a) {}

        void SetMoney(int a)
        {
                m_data = a;
        }

        int GetMoney()
        {
                return m_data;
        }

private:
        int m_data;
};

class Name
{
public:
        Name(string name): m_name(name) {}

        const string &GetName() const
        {
                return m_name;
        }

        friend ostream &operator<<(ostream& out, Name name)
        {
                out << name.GetName();
                return out;
        }
```

```
private:
        string m_name;
};

struct NameLess
{
        bool operator()(const Name &lhs, const Name &rhs) const
        {
                return (lhs.GetName() < rhs.GetName());
        }
};

int main()
{
        //sample 1
        cout << "_________________Iterator with
 int_____________________________________" << endl;
        aggregate<int> agg;

        for (int i = 0; i < 10; i++)
                agg.add(i);

        Iterator< int,aggregate<int> > *it = agg.create_iterator();
        for(it->first(); !it->isDone(); it->next())
                cout << *it->current() << endl;

        //sample 2
        aggregate<Money> agg2;
        Money a(100), b(1000), c(10000);
        agg2.add(a);
        agg2.add(b);
        agg2.add(c);

        cout << "_________________Iterator with Class
 Money___________________________________" << endl;
        Iterator<Money, aggregate<Money> > *it2 = agg2.create_iterator();
        for (it2->first(); !it2->isDone(); it2->next())
                cout << it2->current()->GetMoney() << endl;

        //sample 3
        cout << "_________________Set Iterator with Class
 Name____________________________________" << endl;

        aggregateSet<Name, NameLess> aset;
        aset.add(Name("Qmt"));
        aset.add(Name("Bmt"));
        aset.add(Name("Cmt"));
        aset.add(Name("Amt"));

        setIterator<Name, NameLess, aggregateSet<Name, NameLess> > *it3 =
 aset.create_iterator();
        for (it3->first(); !it3->isDone(); it3->next())
                cout << (*it3->current()) << endl;
}
```

Console output:

```
_________________Iterator with int_____________________________________
```

```
0
1
2
3
4
5
6
7
8
9
_________________Iterator with Class Money_____________________________
100
1000
10000
_________________Set Iterator with Class Name_____________________________
Amt
Bmt
Cmt
Qmt
```

Mediator

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Memento

Without violating encapsulation the Memento Pattern will capture and externalize an object's internal state so that the object can be restored to this state later. Though the GANG OF FOUR[26] uses friend as a way to implement this pattern it is not the best design. It can also be implemented using PIMPL (POINTER TO IMPLEMENTATION OR OPAQUE POINTER)[27]. Best Use case is 'Undo-Redo' in an editor.

26 HTTP://EN.WIKIPEDIA.ORG/WIKI/DESIGN%20PATTERNS
27 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPAQUE%20POINTER

The Originator (the object to be saved) creates a snap-shot of itself as a Memento object, and passes that reference to the Caretaker object. The Caretaker object keeps the Memento until such a time as the Originator may want to revert to a previous state as recorded in the Memento object.

Observer

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Problem

In one place or many places in the application we need to be aware about a system event or an application state change. We'd like to have a standard way of subscribing to listening for system events and a standard way of notifying the interested parties. The notification should be automated after an interested party subscribed to the system event or application state change. There also should be a way to unsubscribe.

Forces

Observers and observables probably should be represented by objects. The observer objects will be notified by the observable objects.

Solution

After subscribing the listening objects will be notified by a way of method call.

```
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

// The Abstract Observer
class ObserverBoardInterface
{
public:
    virtual void update(float a,float b,float c) = 0;
};

// Abstract Interface for Displays
class DisplayBoardInterface
{
public:
    virtual void show() = 0;
};

// The Abstract Subject
```

```
class WeatherDataInterface
{
public:
    virtual void registerOb(ObserverBoardInterface* ob) = 0;
    virtual void removeOb(ObserverBoardInterface* ob) = 0;
    virtual void notifyOb() = 0;
};

// The Concrete Subject
class ParaWeatherData: public WeatherDataInterface
{
public:
    void SensorDataChange(float a,float b,float c)
    {
        m_humidity = a;
        m_temperature = b;
        m_pressure = c;
        notifyOb();
    }

    void registerOb(ObserverBoardInterface* ob)
    {
        m_obs.push_back(ob);
    }

    void removeOb(ObserverBoardInterface* ob)
    {
        m_obs.remove(ob);
    }
protected:
    void notifyOb()
    {
        list<ObserverBoardInterface*>::iterator pos = m_obs.begin();
        while (pos != m_obs.end())
        {
            ((ObserverBoardInterface*
 )(*pos))->update(m_humidity,m_temperature,m_pressure);
            (dynamic_cast<DisplayBoardInterface*>(*pos))->show();
            ++pos;
        }
    }

private:
    float        m_humidity;
    float        m_temperature;
    float        m_pressure;
    list<ObserverBoardInterface* > m_obs;
};

// A Concrete Observer
class CurrentConditionBoard : public ObserverBoardInterface, public
 DisplayBoardInterface
{
public:
    CurrentConditionBoard(ParaWeatherData& a):m_data(a)
    {
        m_data.registerOb(this);
    }
```

```
    void show()
    {
        cout<<"_____CurrentConditionBoard_____"<<endl;
        cout<<"humidity: "<<m_h<<endl;
        cout<<"temperature: "<<m_t<<endl;
        cout<<"pressure: "<<m_p<<endl;
        cout<<"_____"<<endl;
    }

    void update(float h, float t, float p)
    {
        m_h = h;
        m_t = t;
        m_p = p;
    }

private:
    float m_h;
    float m_t;
    float m_p;
    ParaWeatherData& m_data;
};

// A Concrete Observer
class StatisticBoard : public ObserverBoardInterface, public
 DisplayBoardInterface
{
public:
    StatisticBoard(ParaWeatherData&
 a):m_maxt(-1000),m_mint(1000),m_avet(0),m_count(0),m_data(a)
    {
        m_data.registerOb(this);
    }

    void show()
    {
        cout<<"________StatisticBoard_________"<<endl;
        cout<<"lowest  temperature: "<<m_mint<<endl;
        cout<<"highest temperature: "<<m_maxt<<endl;
        cout<<"average temperature: "<<m_avet<<endl;
        cout<<"_____"<<endl;
    }

    void update(float h, float t, float p)
    {
        ++m_count;
        if (t>m_maxt)
        {
            m_maxt = t;
        }
        if (t<m_mint)
        {
            m_mint = t;
        }
        m_avet = (m_avet * (m_count-1) + t)/m_count;
    }

private:
```

```
    float m_maxt;
    float  m_mint;
    float m_avet;
    int m_count;
    ParaWeatherData& m_data;
};


int main(int argc, char *argv[])
{

    ParaWeatherData * wdata = new ParaWeatherData;
    CurrentConditionBoard* currentB = new CurrentConditionBoard(*wdata);
    StatisticBoard* statisticB = new StatisticBoard(*wdata);

    wdata->SensorDataChange(10.2, 28.2, 1001);
    wdata->SensorDataChange(12, 30.12, 1003);
    wdata->SensorDataChange(10.2, 26, 806);
    wdata->SensorDataChange(10.3, 35.9, 900);

    wdata->removeOb(currentB);

    wdata->SensorDataChange(100, 40, 1900);

    delete statisticB;
    delete currentB;
    delete wdata;

    return 0;
}
```

State

The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear as having changed its class.

Strategy

Defines a family of algorithms, encapsulates each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients who use it.

```
#include <iostream>
using namespace std;

class StrategyInterface
{
    public:
        virtual void execute() const = 0;
};

class ConcreteStrategyA: public StrategyInterface
```

```
{
    public:
        virtual void execute() const
        {
            cout << "Called ConcreteStrategyA execute method" << endl;
        }
};

class ConcreteStrategyB: public StrategyInterface
{
    public:
        virtual void execute() const
        {
            cout << "Called ConcreteStrategyB execute method" << endl;
        }
};

class ConcreteStrategyC: public StrategyInterface
{
    public:
        virtual void execute() const
        {
            cout << "Called ConcreteStrategyC execute method" << endl;
        }
};

class Context
{
    private:
        StrategyInterface * strategy_;

    public:
        explicit Context(StrategyInterface *strategy):strategy_(strategy)
        {
        }

        void set_strategy(StrategyInterface *strategy)
        {
            strategy_ = strategy;
        }

        void execute() const
        {
            strategy_->execute();
        }
};

int main(int argc, char *argv[])
{
    ConcreteStrategyA concreteStrategyA;
    ConcreteStrategyB concreteStrategyB;
    ConcreteStrategyC concreteStrategyC;

    Context contextA(&concreteStrategyA);
    Context contextB(&concreteStrategyB);
    Context contextC(&concreteStrategyC);

    contextA.execute(); // output: "Called ConcreteStrategyA execute method"
```

```
    contextB.execute(); // output: "Called ConcreteStrategyB execute method"
    contextC.execute(); // output: "Called ConcreteStrategyC execute method"

    contextA.set_strategy(&concreteStrategyB);
    contextA.execute(); // output: "Called ConcreteStrategyB execute method"
    contextA.set_strategy(&concreteStrategyC);
    contextA.execute(); // output: "Called ConcreteStrategyC execute method"

    return 0;
}
```

Template Method

By defining a skeleton of an algorithm in an operation, deferring some steps to subclasses, the Template Method lets subclasses redefine certain steps of that algorithm without changing the algorithms structure.

Visitor

The Visitor Pattern will represent an operation to be performed on the elements of an object structure by letting you define a new operation without changing the classes of the elements on which it operates.

```
#include <string>
#include <iostream>
#include <vector>

using namespace std;

class Wheel;
class Engine;
class Body;
class Car;

// interface to all car 'parts'
struct CarElementVisitor
{
  virtual void visit(Wheel& wheel) const = 0;
  virtual void visit(Engine& engine) const = 0;
  virtual void visit(Body& body) const = 0;

  virtual void visitCar(Car& car) const = 0;
  virtual ~CarElementVisitor() {}
};

// interface to one part
struct CarElement
{
  virtual void accept(const CarElementVisitor& visitor) = 0;
  virtual ~CarElement() {}
```

```
};

// wheel element, there are four wheels with unique names
class Wheel : public CarElement
{
public:
  explicit Wheel(const string& name) :
    name_(name)
  {
  }
  const string& getName() const
  {
    return name_;
  }
  void accept(const CarElementVisitor& visitor)
  {
    visitor.visit(*this);
  }
private:
    string name_;
};

// engine
class Engine : public CarElement
{
public:
  void accept(const CarElementVisitor& visitor)
  {
    visitor.visit(*this);
  }
};

// body
class Body : public CarElement
{
public:
  void accept(const CarElementVisitor& visitor)
  {
    visitor.visit(*this);
  }
};

// car, all car elements(parts) together
class Car
{
public:
  vector<CarElement*>& getElements()
  {
    return elements_;
  }
  Car()
  {
    // assume that neither push_back nor Wheel(const string&) may throw
    elements_.push_back( new Wheel("front left") );
    elements_.push_back( new Wheel("front right") );
    elements_.push_back( new Wheel("back left") );
    elements_.push_back( new Wheel("back right") );
    elements_.push_back( new Body() );
```

```
    elements_.push_back( new Engine() );
  }
  ~Car()
  {
    for(vector<CarElement*>::iterator it = elements_.begin();
      it != elements_.end(); ++it)
    {
      delete *it;
    }
  }
private:
  vector<CarElement*> elements_;
};

// PrintVisitor and DoVisitor show by using a different implementation the Car
 class is unchanged
// even though the algorithm is different in PrintVisitor and DoVisitor.
class CarElementPrintVisitor : public CarElementVisitor
{
public:
  void visit(Wheel& wheel) const
  {
    cout << "Visiting " << wheel.getName() << " wheel" << endl;
  }
  void visit(Engine& engine) const
  {
    cout << "Visiting engine" << endl;
  }
  void visit(Body& body) const
  {
    cout << "Visiting body" << endl;
  }
  void visitCar(Car& car) const
  {
    cout << endl << "Visiting car" << endl;
    vector<CarElement*>& elems = car.getElements();
    for(vector<CarElement*>::iterator it = elems.begin();
      it != elems.end(); ++it )
    {
      (*it)->accept(*this);        // this issues the callback i.e. to this from the
 element
    }
    cout << "Visited car" << endl;
  }
};

class CarElementDoVisitor : public CarElementVisitor
{
public:
  // these are specific implementations added to the original object without
 modifying the original struct
  void visit(Wheel& wheel) const
  {
    cout << "Kicking my " << wheel.getName() << " wheel" << endl;
  }
  void visit(Engine& engine) const
  {
    cout << "Starting my engine" << endl;
```

```
  }
  void visit(Body& body) const
  {
    cout << "Moving my body" << endl;
  }
  void visitCar(Car& car) const
  {
    cout << endl << "Starting my car" << endl;
    vector<CarElement*>& elems = car.getElements();
    for(vector<CarElement*>::iterator it = elems.begin();
      it != elems.end(); ++it )
    {
      (*it)->accept(*this);        // this issues the callback i.e. to this from the
 element
    }
    cout << "Stopped car" << endl;
  }
};

int main()
{
  Car car;
  CarElementPrintVisitor printVisitor;
  CarElementDoVisitor doVisitor;

  printVisitor.visitCar(car);
  doVisitor.visitCar(car);

  return 0;
}
```

Model-View-Controller (MVC)

A pattern often used by applications that need the ability to maintain multiple views of the same data. The model-view-controller pattern was until recently a very common pattern especially for graphic user interlace programming, it splits the code in 3 pieces. The model, the view, and the controller.

The Model is the actual data representation (for example, Array vs Linked List) or other objects representing a database. The View is an interface to reading the model or a fat client GUI. The Controller provides the interface of changing or modifying the data, and then selecting the "Next Best View" (NBV).

Newcomers will probably see this "MVC" model as wasteful, mainly because you are working with many extra objects at runtime, when it seems like one giant object will do. But the secret to the MVC pattern is not writing the code, but in maintaining it, and allowing people to modify the code without changing much else. Also, keep in mind, that different developers have different strengths and weaknesses, so team building around MVC is easier. Imagine a View Team that is responsible for great views, a Model Team that knows a lot about data, and a Controller Team that

see the big picture of application flow, handing requests, working with the model, and selecting the most appropriate next view for that client.

For example: A naive central database can be organized using only a "model", for example, a straight array. However, later on, it may be more applicable to use a linked list. All array accesses will have to be remade into their respective Linked List form (for example, you would change myarray[5] into mylist.at(5) or whatever is equivalent in the language you use).

Well, if we followed the MVC pattern, the central database would be accessed using some sort of a function, for example, myarray.at(5). If we change the model from an array to a linked list, all we have to do is change the view with the model, and the whole program is changed. Keep the interface the same but change the underpinnings of it. This would allow us to make optimizations more freely and quickly than before.

One of the great advantages of the Model-View-Controller Pattern is obviously the ability to reuse the application's logic (which is implemented in the model) when implementing a different view. A good example is found in web development, where a common task is to implement an external API inside of an existing piece of software. If the MVC pattern has cleanly been followed, this only requires modification to the controller, which can have the ability to render different types of views dependent on the content type requested by the user agent.

28

29

6.4 Libraries

Libraries allow existing code to be reused in a program. Libraries are like programs except that instead of relying on **main()** to do the work you call the specific functions provided by the library to do the work. Functions provide the interface between the program being written and the library being used. This interface is called *Application Programming Interface*[30] or API.

Libraries should and tend to be domain specific as to permit greater mobility across applications, and provide extended specialization. Libraries that are not, are often

28 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

29 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

30 HTTP://EN.WIKIPEDIA.ORG/WIKI/APPLICATION%20PROGRAMMING%20INTERFACE

header only distribution, intended for static linking as to permit the compiler and the application, only to use the needed bits of code.

What is an API?

To a programmer, an operating system is defined by its *API*. *API* stands for *Application Programming Interface*. An *API* encompasses all the function calls that an application program can communicate with the hardware or the operating system, or any other application that provides a set of interfaces to the programmer (i.e.: a library), as well as definitions of associated data types and structures. Most *APIs* are defined on the application *Software Development Kit* (SDK) for program development.

In simple terms the *API* can be considered as the interface through which the user (or user programs) will be able interact with the operating system, hardware or other programs to make them to perform a task that may also result in obtaining a result message.

Can an API be called a *framework*?

No, a *framework* may provide an API, but a *framework* is more than a simple API. By default a framework also defines how the code is written, it is a set of solutions, even classes, that as a group addresses the handling of a limited set of related problems and provides not only an API but a default functionality, well designed *frameworks* enable its interchangeability for a similar *framework*, striving to provides the same API.

31

As seen in the FILE ORGANIZATION SECTION[32], compiled libraries consists in C++ headers files that are included by the preprocessor and binary library files which are used by the linker to generate the resulting compilation. For a dynamically linked library, only the loading code is added to the compilation that uses them, the actual loading of the library is done in the memory at run-time.

Programs can make use of libraries in two forms, as static or dynamic depending on how the programmer decides to distribute its code or even due to the licensing

31 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
32 Chapter 3.1.5 on page 49

used by third party libraries, the STATIC AND DYNAMIC LIBRARIES[33] section of this book will cover in depth this subject.

Note:
As we will see when covering MULTI-THREADING[a] when selecting libraries. Remember to verify if they conform to the your requirements on that area.

a Chapter 6.6.2 on page 606

6.4.1 Third party libraries

Additional functionality that goes beyond the standard libraries (like GARBAGE COLLECTION[34]) are available (often free) by third party libraries, but remember that third party libraries do not necessarily provide the same ubiquitous cross-platform functionality or an API style conformant with as standard libraries. The main motivation for their existence is for preventing one tho reinvent the wheel and to make efforts converge; too much energy has been spent by generations of programmers to write safe and "portable" code.

There are several libraries a the programmer is expected to know about or have at least a passing idea of what they are. Time, consistency and extended references will make a few libraries pop-out from the rest. One notable example is the highly respected collection of BOOST LIBRARIES[35] that we will examine ahead.

Licensing on third party libraries

The programmer may also be limited by the requirements of the license used on external libraries that he has no direct control, for instance the use of the GNU GENERAL PUBLIC LICENSE[36] (GNU GPL) code in closed source applications isn't permitted to address this issue the FSF provides an alternative in the form of the GNU LGPL license that permits such uses but only in the dynamically linked form, this is mirrored by several other legal requirements a programmer must attend and comply to.

33 Chapter 6.4.1 on page 586
34 Chapter 6.1 on page 540
35 Chapter 6.4.2 on page 588
36 HTTP://EN.WIKIPEDIA.ORG/WIKI/GNU%20GENERAL%20PUBLIC%20LICENSE

6.4.2 Static and Dynamic Libraries

Libraries come in two forms, either in *source form* or in *compiled/binary form*. Libraries in source-form must first be compiled before they can be included in another project. This will transform the libraries' cpp-files into a lib-file. If a program must be recompiled to run with a new version of a library, but does not need any further changes, the library is said to be *source compatible*. If a program does not need to be modified and recompiled to use a new version of a library, the library is then classified as being *binary compatible*.

Advantages of using static binaries:

- Simplification of program distribution (fewer files).
- Code simplification (no version checks as required in dynamic libraries).
- Will only compile the code that is used.

Disadvantages of using static binaries:

- Waste of resources: Generates larger binaries, since the library is compiled into the executable. Wastes memory as the library cannot be shared (in memory) between processes (depending on the operating system).
- Program will not benefit from bug fixes or extensions in the libraries without being recompiled.

Binary/Source Compatibility of libraries

A library is said to be **binary compatible** if the program that dynamically links to an earlier version of that library, continues to work using another versions of the same library. If a recompilation of the program is needed for it to run with each new version the library is said to be **source compatible**.

Producing binary compatible libraries is beneficial for distribution but harder to maintain by the programmer. It is often seen as a better solution to do static linking, if the library is only source compatible, since it will not cause problems to the end-user.

Binary compatibility saves a lot of trouble and is a signal that the library reached a status of stability. It makes it easier to distribute software for a certain platform. Without ensuring binary compatibility between releases, people will be forced to offer statically linked binaries.

header-only libraries

Another distinction that is commonly made about libraries are on how they are distributed (regarding structure and use). A library that is contained only on header files is considered header-only library. Often this means that they are simpler and easy to use, however this will not be the ideal solution for complex code, it will not only hamper readability but result in larger compile times. Also depending on the compiler and it's optimizing capabilities (or options) can, due to the resulting inlining, generate larger binaries. This may not be as important in libraries mostly implemented with templates. Header-only libraries will always contain the source code to the implementation, commercial is rare.

37

38

6.5 Boost Library

The **Boost library**[39] (HTTP://WWW.BOOST.ORG/)[40] provides free PEER-REVIEWED[41] , OPEN SOURCE[42] LIBRARIES[43] that extend the functionality of C++. Most of the libraries are licensed under the BOOST SOFTWARE LICENSE[44], designed to allow Boost to be used with both open and CLOSED SOURCE[45] projects.

Many of Boost's founders are on the C++ STANDARD[46] committee and several Boost libraries have been accepted for incorporation into the TECHNICAL REPORT 1[47] of C++0X[48]. Although Boost was begun by members of the C++ Standards Committee Library Working Group, participation has expanded to include thousands of programmers from the C++ community at large.

The emphasis is on libraries which work well with the C++ Standard Library. The libraries are aimed at a wide range of C++ users and application domains, and are

37 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
38 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
39 HTTP://EN.WIKIPEDIA.ORG/WIKI/BOOST%20%28PROGRAMMING%29
40 HTTP://WWW.BOOST.ORG/)
41 HTTP://EN.WIKIPEDIA.ORG/WIKI/PEER-REVIEW
42 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPEN%20SOURCE
43 HTTP://EN.WIKIPEDIA.ORG/WIKI/LIBRARY%20%28COMPUTER%20SCIENCE%29
44 HTTP://WWW.BOOST.ORG/MORE/LICENSE_INFO.HTML
45 HTTP://EN.WIKIPEDIA.ORG/WIKI/CLOSED%20SOURCE
46 HTTP://EN.WIKIPEDIA.ORG/WIKI/ISO%2FIEC%2014882
47 HTTP://EN.WIKIPEDIA.ORG/WIKI/TECHNICAL%20REPORT%201
48 HTTP://EN.WIKIPEDIA.ORG/WIKI/C%2B%2B0X

in regular use by thousands of programmers. They range from general-purpose libraries like SMARTPTR[49], to OS Abstractions like FILESYSTEM[50], to libraries primarily aimed at other library developers and advanced C++ users, like MPL[51].

A further goal is to establish "existing practice" and provide reference implementations so that Boost libraries are suitable for eventual standardization. Ten Boost libraries will be included in the C++ STANDARDS COMMITTEE'S[52] upcoming C++ STANDARD LIBRARY TECHNICAL REPORT[53] as a step toward becoming part of a future C++ Standard.

In order to ensure efficiency and flexibility, Boost makes extensive use of TEMPLATE[54]s. Boost has been a source of extensive work and research into GENERIC PROGRAMMING[55] and METAPROGRAMMING[56] in C++.

6.5.1 extension libraries

- Algorithms
- Concurrent programming (THREADS[57])
- CONTAINERS[58]
  - ARRAY[59] - Management of fixed-size arrays with STL container semantics
  - BOOST GRAPH LIBRARY (BGL)[60] - Generic graph containers, components and algorithms
  - MULTI-ARRAY[61] - Simplifies creation of N-dimensional arrays
  - MULTI-INDEX CONTAINERS[62] - Containers with built in indexes that allow different sorting and access semantics
  - POINTER CONTAINERS[63] - Containers modeled after most standard STL containers that allow for transparent management of pointers to values

49 HTTP://WWW.BOOST.ORG/LIBS/SMART_PTR
50 HTTP://WWW.BOOST.ORG/LIBS/FILESYSTEM
51 HTTP://WWW.BOOST.ORG/LIBS/MPL
52 HTTP://WWW.OPEN-STD.ORG/JTC1/SC22/WG21/
53 HTTP://STD.DKUUG.DK/JTC1/SC22/WG21/DOCS/LIBRARY_TECHNICAL_REPORT.HTML
54 HTTP://EN.WIKIPEDIA.ORG/WIKI/TEMPLATE%20%28PROGRAMMING%29
55 HTTP://EN.WIKIPEDIA.ORG/WIKI/GENERIC%20PROGRAMMING
56 HTTP://EN.WIKIPEDIA.ORG/WIKI/METAPROGRAMMING
57 HTTP://EN.WIKIPEDIA.ORG/WIKI/THREAD%20%28COMPUTER%20SCIENCE%29
58 HTTP://EN.WIKIPEDIA.ORG/WIKI/DATA%20STRUCTURE
59 HTTP://BOOST.ORG/DOC/HTML/ARRAY.HTML
60 HTTP://BOOST.ORG/LIBS/GRAPH/DOC/TABLE_OF_CONTENTS.HTML
61 HTTP://BOOST.ORG/LIBS/MULTI_ARRAY/DOC/INDEX.HTML
62 HTTP://BOOST.ORG/LIBS/MULTI_INDEX/DOC/INDEX.HTML
63 HTTP://BOOST.ORG/LIBS/PTR_CONTAINER/DOC/PTR_CONTAINER.HTML

  - PROPERTY MAP[64] - Interface specifications in the form of concepts and a general purpose interface for mapping key values to objects
  - VARIANT[65] - A safe and generic stack-based object container that allows for the efficient storage of and access to an object of a type that can be chosen from among a set of types that must be specified at compile time.
- Correctness and TESTING[66]
  - CONCEPT CHECK[67] - Allows for the enforcement of actual template parameter requirements (concepts)
  - STATIC ASSERT[68] - Compile time assertion support
  - BOOST TEST LIBRARY[69] - A matched set of components for writing test programs, organizing tests into test cases and test suites, and controlling their runtime execution
- Data structures
  - DYNAMIC_BITSET[70] - Dynamic `std::bitset`-like data structure
- Function objects and HIGHER-ORDER PROGRAMMING[71]
  - BIND[72] and MEM_FN[73] - General binders for functions, function objects, function pointers and member functions
  - FUNCTION[74] - Function object wrappers for deferred calls. Also, provides a generalized mechanism for callbacks
  - FUNCTIONAL[75] - Enhancements to the function object adapters specified in the C++ Standard Library, including:
    - FUNCTION OBJECT TRAITS[76]
    - NEGATORS[77]
    - BINDERS[78]
    - ADAPTERS FOR POINTERS TO FUNCTIONS[79]

64 HTTP://BOOST.ORG/LIBS/PROPERTY_MAP/PROPERTY_MAP.HTML
65 HTTP://BOOST.ORG/DOC/HTML/VARIANT.HTML
66 HTTP://EN.WIKIPEDIA.ORG/WIKI/SOFTWARE%20TESTING
67 HTTP://BOOST.ORG/LIBS/CONCEPT_CHECK/CONCEPT_CHECK.HTM
68 HTTP://BOOST.ORG/DOC/HTML/BOOST_STATICASSERT.HTML
69 HTTP://BOOST.ORG/LIBS/TEST/DOC/INDEX.HTML
70 HTTP://BOOST.ORG/LIBS/DYNAMIC_BITSET/
71 HTTP://EN.WIKIPEDIA.ORG/WIKI/HIGHER-ORDER%20PROGRAMMING
72 HTTP://BOOST.ORG/LIBS/BIND/BIND.HTML
73 HTTP://WWW.BOOST.ORG/LIBS/BIND/MEM_FN.HTML
74 HTTP://BOOST.ORG/DOC/HTML/FUNCTION.HTML
75 HTTP://BOOST.ORG/LIBS/FUNCTIONAL/INDEX.HTML
76 HTTP://BOOST.ORG/LIBS/FUNCTIONAL/FUNCTION_TRAITS.HTML
77 HTTP://BOOST.ORG/LIBS/FUNCTIONAL/NEGATORS.HTML
78 HTTP://BOOST.ORG/LIBS/FUNCTIONAL/BINDERS.HTML
79 HTTP://BOOST.ORG/LIBS/FUNCTIONAL/PTR_FUN.HTML

    - ADAPTERS FOR POINTERS TO MEMBER FUNCTIONS[80]
  - HASH[81] - An implementation of the hash function object specified by the C++ Technical Report 1 (TR1). Can be used as the default hash function for unordered associative containers
  - LAMBDA[82] - In the spirit of LAMBDA ABSTRACTIONS[83], allows for the definition of small anonymous function objects and operations on those objects at a call site, using placeholders, especially for use with deferred callbacks from algorithms.
  - REF[84] - Provides utility class templates for enhancing the capabilities of standard C++ references, especially for use with generic functions
  - RESULT_OF[85] - Helps in the determination of the type of a call expression
  - SIGNALS AND SLOTS[86] - Managed signals and slots callback implementation
- GENERIC PROGRAMMING[87]
- GRAPHS[88]
- Input/output
- Interlanguage support (for PYTHON[89])
- ITERATORS[90]
  - ITERATORS[91]
  - OPERATORS[92] - Class templates that help with overloaded operator definitions for user defined iterators and classes that can participate in arithmetic computation.
  - TOKENIZER[93] - Provides a view of a set of tokens contained in a sequence that makes them appear as a container with iterator access
- Math and Numerics
- MEMORY[94]

80 HTTP://BOOST.ORG/LIBS/FUNCTIONAL/MEM_FUN.HTML
81 HTTP://BOOST.ORG/DOC/HTML/HASH.HTML
82 HTTP://BOOST.ORG/DOC/HTML/LAMBDA.HTML
83 HTTP://EN.WIKIPEDIA.ORG/WIKI/LAMBDA%20CALCULUS
84 HTTP://BOOST.ORG/DOC/HTML/REF.HTML
85 HTTP://BOOST.ORG/LIBS/UTILITY/UTILITY.HTM#RESULT_OF
86 HTTP://BOOST.ORG/DOC/HTML/SIGNALS.HTML
87 HTTP://EN.WIKIPEDIA.ORG/WIKI/GENERIC%20PROGRAMMING
88 HTTP://EN.WIKIPEDIA.ORG/WIKI/GRAPH%20%28DATA%20STRUCTURE%29
89 HTTP://EN.WIKIPEDIA.ORG/WIKI/PYTHON%20%28PROGRAMMING%20LANGUAGE%29
90 HTTP://EN.WIKIPEDIA.ORG/WIKI/ITERATOR%23C.2B.2B
91 HTTP://BOOST.ORG/LIBS/ITERATOR/DOC/INDEX.HTML
92 HTTP://BOOST.ORG/LIBS/UTILITY/OPERATORS.HTM
93 HTTP://BOOST.ORG/LIBS/TOKENIZER/INDEX.HTML
94 HTTP://EN.WIKIPEDIA.ORG/WIKI/MAIN%20MEMORY

- POOL[95] - Provides a simple segregated storage based memory management scheme
- SMART_PTR[96] - A collection of smart pointer class templates with different pointee management semantics
  - SCOPED_PTR[97] - Owns the pointee (single object)
  - SCOPED_ARRAY[98] - Like scoped_ptr, but for arrays
  - SHARED_PTR[99] - Potentially shares the pointer with other shared_ptrs. Pointee is destroyed when last shared_ptr to it is destroyed
  - SHARED_ARRAY[100] - Like shared_ptr, but for arrays
  - WEAK_PTR[101] - Provides a "weak" reference to an object that is already managed by a shared_ptr
  - INTRUSIVE_PTR[102] - Similared to shared_ptr, but uses a reference count provided by the pointee
- UTILITY[103] - Miscellaneous support classes, including:
  - BASE FROM MEMBER IDIOM[104] - Provides a workaround for a class that needs to initialize a member of a base class inside its own (i.e., the derived class') constructor's initializer list
  - CHECKED DELETE[105] - Check if an attempt is made to destroy an object or array of objects using a pointer to an incomplete type
  - NEXT AND PRIOR FUNCTIONS[106] - Allow for easier motion of a forward or bidirectional iterator, especially when the results of such a motion need to be stored in a separate iterator (i.e., should not change the original iterator)
  - NONCOPYABLE[107] - Allows for the prohibition of copy construction and copy assignment
  - ADDRESSOF[108] - Allows for the acquisition of an object's real address, bypassing any overloads of operator&(), in the process
  - RESULT_OF[109] - Helps in the determination of the type of a call expression

95 HTTP://BOOST.ORG/LIBS/POOL/DOC/INDEX.HTML
96 HTTP://BOOST.ORG/LIBS/SMART_PTR/SMART_PTR.HTM
97 HTTP://BOOST.ORG/LIBS/SMART_PTR/SCOPED_PTR.HTM
98 HTTP://BOOST.ORG/LIBS/SMART_PTR/SCOPED_ARRAY.HTM
99 HTTP://BOOST.ORG/LIBS/SMART_PTR/SHARED_PTR.HTM
100 HTTP://BOOST.ORG/LIBS/SMART_PTR/SHARED_ARRAY.HTM
101 HTTP://BOOST.ORG/LIBS/SMART_PTR/WEAK_PTR.HTM
102 HTTP://BOOST.ORG/LIBS/SMART_PTR/INTRUSIVE_PTR.HTML
103 HTTP://BOOST.ORG/LIBS/UTILITY/UTILITY.HTM
104 HTTP://BOOST.ORG/LIBS/UTILITY/BASE_FROM_MEMBER.HTML
105 HTTP://BOOST.ORG/LIBS/UTILITY/CHECKED_DELETE.HTML
106 HTTP://BOOST.ORG/LIBS/UTILITY/UTILITY.HTM#FUNCTIONS_NEXT_PRIOR
107 HTTP://BOOST.ORG/LIBS/UTILITY/UTILITY.HTM#CLASS_NONCOPYABLE
108 HTTP://BOOST.ORG/LIBS/UTILITY/UTILITY.HTM#ADDRESSOF
109 HTTP://BOOST.ORG/LIBS/UTILITY/UTILITY.HTM#RESULT_OF

- Miscellaneous
- PARSERS[110]
- Preprocessor metaprogramming
- STRING[111] and text processing
  - LEXICAL_CAST[112] - Type conversions to/from text
  - FORMAT[113] - Type safe argument formatting according to a format string
  - IOSTREAMS[114] - C++ streams and stream buffer assistance for new sources/sinks, filters framework
  - REGEX[115] - Support for regular expressions
  - SPIRIT[116] - An object-oriented recursive-descent parser generator framework
  - STRING ALGORITHMS[117] - A collection of various algorithms related to strings
  - TOKENIZER[118] - Allows for the partitioning of a string or other character sequence into TOKEN[119]s
  - WAVE[120] - Standards conformant implementation of the mandated C99[121] / C++ pre-processor functionality packed behind an easy to use interface
- TEMPLATE METAPROGRAMMING[122]
  - MPL[123] - A general purpose high-level metaprogramming framework of compile-time algorithms, sequences and metafunctions
  - STATIC ASSERT[124] - Compile time assertion support
  - TYPE TRAITS[125] - Templates that define the fundamental properties of types
- Workarounds for broken compilers

The current Boost release contains 87 individual libraries, including the following three:

110 HTTP://EN.WIKIPEDIA.ORG/WIKI/SPIRIT%20PARSER%20FRAMEWORK
111 HTTP://EN.WIKIPEDIA.ORG/WIKI/STRING%20%28COMPUTER%20SCIENCE%29
112 HTTP://BOOST.ORG/LIBS/CONVERSION/LEXICAL_CAST.HTM
113 HTTP://BOOST.ORG/LIBS/FORMAT/INDEX.HTML
114 HTTP://BOOST.ORG/LIBS/IOSTREAMS/DOC/INDEX.HTML
115 HTTP://WWW.BOOST.ORG/DOC/LIBS/1_44_0/LIBS/REGEX/DOC/HTML/INDEX.HTML
116 HTTP://EN.WIKIPEDIA.ORG/WIKI/SPIRIT%20PARSER%20FRAMEWORK
117 HTTP://BOOST.ORG/DOC/HTML/STRING_ALGO.HTML
118 HTTP://BOOST.ORG/LIBS/TOKENIZER/INDEX.HTML
119 HTTP://EN.WIKIPEDIA.ORG/WIKI/TOKEN%20%28PARSER%29
120 HTTP://BOOST.ORG/LIBS/WAVE/INDEX.HTML
121 HTTP://EN.WIKIPEDIA.ORG/WIKI/C99
122 HTTP://EN.WIKIPEDIA.ORG/WIKI/TEMPLATE%20METAPROGRAMMING
123 HTTP://BOOST.ORG/LIBS/MPL/DOC/INDEX.HTML
124 HTTP://BOOST.ORG/DOC/HTML/BOOST_STATICASSERT.HTML
125 HTTP://BOOST.ORG/DOC/HTML/BOOST_TYPETRAITS.HTML

6.5.2 noncopyable

The `boost::noncopyable` utility class that ENSURES THAT OBJECTS OF A CLASS ARE NEVER COPIED[126].

```
class C : boost::noncopyable
{
  ...
};
```

6.5.3 Linear algebra – uBLAS

Boost includes the **uBLAS** LINEAR ALGEBRA[127] library, with BLAS[128] support for vectors and matrices. uBlas supports a wide range of linear algebra operations, and has bindings to some widely used numerics libraries, such as ATLAS, BLAS and LAPACK.

- Example showing how to multiply a vector with a matrix:

```
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
#include <iostream>

using namespace boost::numeric::ublas;

/* "y = Ax" example */
int main ()
{
      vector<double> x(2);
      x(0) = 1; x(1) = 2;

      matrix<double> A(2,2);
      A(0,0) = 0; A(0,1) = 1;
      A(1,0) = 2; A(1,1) = 3;

      vector<double> y = prod(A, x);

      std::cout << y << std::endl;
      return 0;
}
```

126 Chapter 4.3.1 on page 394

127 HTTP://EN.WIKIPEDIA.ORG/WIKI/LINEAR%20ALGEBRA

128 HTTP://EN.WIKIPEDIA.ORG/WIKI/BASIC%20LINEAR%20ALGEBRA%20SUBPROGRAMS

6.5.4 Generating random numbers – Boost.Random

Boost provides distribution-independent PSEUDORANDOM NUMBER GENERATOR[129]s and PRNG-independent probability distributions, which are combined to build a concrete generator.

- Example showing how to sample from a NORMAL DISTRIBUTION[130] using the MERSENNE TWISTER[131] generator:

```
#include <boost/random.hpp>
#include <ctime>

using namespace boost;

double SampleNormal (double mean, double sigma)
{
    // Create a Mersenne twister random number generator
    // that is seeded once with #seconds since 1970
    static mt19937 rng(static_cast<unsigned> (std::time(0)));

    // select Gaussian probability distribution
    normal_distribution<double> norm_dist(mean, sigma);

    // bind random number generator to distribution, forming a function
    variate_generator<mt19937&, normal_distribution<double> >
 normal_sampler(rng, norm_dist);

    // sample from the distribution
    return normal_sampler();
}
```

See BOOST RANDOM NUMBER LIBRARY[132] for more details.

6.5.5 Multi-threading – Boost.Thread

Example code that demonstrates creation of threads:

```
#include <boost/thread/thread.hpp>
#include <iostream>

using namespace std;

void hello_world()
{
  cout << "Hello world, I'm a thread!" << endl;
```

129 HTTP://EN.WIKIPEDIA.ORG/WIKI/PSEUDORANDOM%20NUMBER%20GENERATOR
130 HTTP://EN.WIKIPEDIA.ORG/WIKI/NORMAL%20DISTRIBUTION
131 HTTP://EN.WIKIPEDIA.ORG/WIKI/MERSENNE%20TWISTER
132 HTTP://BOOST.ORG/LIBS/RANDOM/

```
}

int main(int argc, char* argv[])
{
  // start two new threads that calls the "hello_world" function
  boost::thread my_thread1(&hello_world);
  boost::thread my_thread2(&hello_world);

  // wait for both threads to finish
  my_thread1.join();
  my_thread2.join();

  return 0;
}
```

See also THREADING WITH BOOST - PART I: CREATING THREADS[133]

6.5.6 Thread locking

Example usage of a mutex to enforce exclusive access to a function:

```
#include <iostream>
#include <boost/thread.hpp>

void locked_function ()
{
    // function access mutex
    static boost::mutex m;
    // wait for mutex lock
    boost::mutex::scoped_lock lock(m);

    // critical section
    // TODO: Do something

    // auto-unlock on return
}

int main (int argc, char* argv[])
{
    locked_function();
    return 0;
}
```

Example of read/write locking of a property:

```
#include <iostream>
#include <boost/thread.hpp>

/** General class for thread-safe properties of any type. */
template <class T>
```

133 HTTP://ANTONYM.ORG/2009/05/THREADING-WITH-BOOST---PART-I-CREATING-THREADS.HTML

```
class lock_prop : boost::noncopyable {
public:
    lock_prop () {}

    /** Set property value. */
    void operator = (const T & v) {
        // wait for exclusive write access
        boost::unique_lock<boost::shared_mutex> lock(mutex);

        value = v;
    }

    /** Get property value. */
    T operator () () const {
        // wait for shared read access
        boost::shared_lock<boost::shared_mutex> lock(mutex);

        return value;
    }

private:
    /// Property value.
    T                           value;
    /// Mutex to restrict access
    mutable boost::shared_mutex mutex;
};

int main () {
    // read/write locking property
    lock_prop<int> p1;
    p1 = 10;
    int a = p1();

    return 0;
}
```

- INTRODUCTION TO BOOST.THREADS[134] in DR. DOBB'S JOURNAL[135]. (2002)
- WHAT'S NEW IN BOOST THREADS?[136] in DR. DOBB'S JOURNAL[137]. (2008)
- Boost.Threads API REFERENCE[138].
- THREADPOOL LIBRARY[139] based on Boost.Thread

140

134 HTTP://WWW.DDJ.COM/DEPT/CPP/184401518
135 HTTP://EN.WIKIPEDIA.ORG/WIKI/DR.%20DOBB%27S%20JOURNAL
136 HTTP://WWW.DDJ.COM/CPP/211600441
137 HTTP://EN.WIKIPEDIA.ORG/WIKI/DR.%20DOBB%27S%20JOURNAL
138 HTTP://WWW.BOOST.ORG/DOC/HTML/THREAD.HTML
139 HTTP://THREADPOOL.SOURCEFORGE.NET
140 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

6.6 Cross-Platform development

The section is to introduce programmer to programming with the aim of portability across several OSs environments. In today's world it does not seem appropriate to constrain applications to a single operating system or computer platform, and there is an increasing need to address programming in a *cross platform* manner.

141

6.6.1 The Windows 32 API

Win32 API is a set of functions defined in the *Windows OS*, in other words it is the **Windows API**, this is the name given by *Microsoft* to the core set of APPLICATION PROGRAMMING INTERFACE[142]s available in the MICROSOFT WINDOWS[143] OPERATING SYSTEMS[144]. It is designed for usage by C[145]/C++[146] programs and is the most direct way to interact with a *Windows* system for SOFTWARE APPLICATIONS[147]. Lower level access to a *Windows* system, mostly required for DEVICE DRIVERS[148], is provided by the WINDOWS DRIVER MODEL[149] in current versions of *Windows*.

One can get more information about the *API* and support from *Microsoft* itself, using the MSDN Library (HTTP://MSDN.MICROSOFT.COM/[150]) essentially a resource for developers using Microsoft tools, products, and technologies. It contains a bounty of technical programming information, including sample code, documentation, technical articles, and reference guides. You can also check out Wikibooks WINDOWS PROGRAMMING[151] book for some more detailed information that goes beyond the scope of this book.

141 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
142 HTTP://EN.WIKIPEDIA.ORG/WIKI/APPLICATION%20PROGRAMMING%20INTERFACE
143 HTTP://EN.WIKIPEDIA.ORG/WIKI/MICROSOFT%20WINDOWS
144 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPERATING%20SYSTEMS
145 HTTP://EN.WIKIPEDIA.ORG/WIKI/C
146 HTTP://EN.WIKIPEDIA.ORG/WIKI/C%20PLUS%20PLUS
147 HTTP://EN.WIKIPEDIA.ORG/WIKI/APPLICATION%20SOFTWARE
148 HTTP://EN.WIKIPEDIA.ORG/WIKI/DEVICE%20DRIVERS
149 HTTP://EN.WIKIPEDIA.ORG/WIKI/WINDOWS%20DRIVER%20MODEL
150 HTTP://MSDN.MICROSOFT.COM/
151 HTTP://EN.WIKIBOOKS.ORG/WIKI/WINDOWS%20PROGRAMMING

A SOFTWARE DEVELOPMENT KIT[152] (SDK) is available for *Windows*, which provides documentation and tools to enable developers to create software using the *Windows API* and associated *Windows* technologies. (HTTP://WWW.MICROSOFT.COM/DOWNLOADS/[153])

History

The Windows API has always exposed a large part of the underlying structure of the various Windows systems for which it has been built to the programmer. This has had the advantage of giving Windows programmers a great deal of flexibility and power over their applications. However, it also has given Windows applications a great deal of responsibility in handling various low-level, sometimes tedious, operations that are associated with a GRAPHICAL USER INTERFACE[154].

CHARLES PETZOLD[155], writer of various well read Windows API books, has said: *"The original hello-world program in the Windows 1.0 SDK was a bit of a scandal. HELLO.C was about 150 lines long, and the HELLO.RC resource script had another 20 or so more lines. (...) Veteran C programmers often curled up in horror or laughter when encountering the Windows hello-world program."*. A HELLO WORLD PROGRAM[156] is a often used programming example, usually designed to show the easiest possible application on a system that can actually do something (i.e. print a line that says "Hello World").

Over the years, various changes and additions were made to the Windows Operating System, and the Windows API changed and grew to reflect this. The windows API for WINDOWS 1.0[157] supported less than 450 FUNCTION CALLS[158], where in modern versions of the Windows API there are thousands. In general, the interface has remained fairly consistent however, and a old Windows 1.0 application will still look familiar to a programmer who is used to the modern Windows API.

A large emphasis has been put by MICROSOFT[159] on maintaining software BACKWARDS COMPATIBILITY[160]. To achieve this, Microsoft sometimes went as far as supporting software that was using the API in a undocumented or even (program-

152 HTTP://EN.WIKIPEDIA.ORG/WIKI/SOFTWARE%20DEVELOPMENT%20KIT
153 HTTP://WWW.MICROSOFT.COM/DOWNLOADS/
154 HTTP://EN.WIKIPEDIA.ORG/WIKI/GRAPHICAL%20USER%20INTERFACE
155 HTTP://EN.WIKIPEDIA.ORG/WIKI/CHARLES%20PETZOLD
156 HTTP://EN.WIKIPEDIA.ORG/WIKI/HELLO%20WORLD%20PROGRAM
157 HTTP://EN.WIKIPEDIA.ORG/WIKI/WINDOWS%201.0
158 HTTP://EN.WIKIPEDIA.ORG/WIKI/SUBROUTINE
159 HTTP://EN.WIKIPEDIA.ORG/WIKI/MICROSOFT
160 HTTP://EN.WIKIPEDIA.ORG/WIKI/BACKWARDS%20COMPATIBILITY

matically) illegal way. RAYMOND CHEN[161], a Microsoft developer who works on the Windows API, has said that he *"could probably write for months solely about bad things apps do and what we had to do to get them to work again (often in spite of themselves). Which is why I get particularly furious when people accuse Microsoft of maliciously breaking applications during OS upgrades. If any application failed to run on Windows 95, I took it as a personal failure."*

Variables and Win32

Win32 uses an extended set of data types, using C's typedef mechanism These include:

- BYTE - `unsigned` 8 bit integer.
- DWORD - 32 bit `unsigned` integer.
- LONG - 32 bit signed integer.
- LPDWORD - 32 bit pointer to DWORD.
- LPCSTR - 32 bit pointer to constant character string.
- LPSTR - 32 bit pointer to character string.
- UINT - 32 bit `unsigned` int.
- WORD - 16 bit `unsigned` int.
- HANDLE - opaque pointer to system data.

Of course standard data types are also available when programming with **Win32 API**.

Windows Libraries (DLLs)

In Windows, library code exists in a number of forms, and can be accessed in various ways.

Normally, the only thing that is needed is to include in the appropriate header file on the source code the information to the compiler, and linking to the .lib file will occur during the linking phase.

This .lib file either contains code which is to be statically linked into compiled object code or contains code to allow access to a dynamically link to a binary library(.DLL) on the system.

It is also possible to generate a binary library .DLL within C++ by including appropriate information such as an import/export table when compiling and linking.

161 HTTP://EN.WIKIPEDIA.ORG/WIKI/RAYMOND%20CHEN

DLLs stand for Dynamic Link Libraries, the basic file of functions that are used in some programs. Many newer C++ IDEs such as Dev-CPP support such libraries.

Common libraries on Windows include those provided by the Platform Software Development Kit, Microsoft Foundation Class and a C++ interface to .Net Framework assemblies.

Although not strictly use as library code, the Platform SDK and other libraries provide a set of standardized interfaces to objects accessible via the COMPONENT OBJECT MODEL[162] implemented as part of Windows.

API conventions and Win32 API Functions (by focus)

Time

Time measurement has to come from the OS in relation to the hardware it is run, unfortunately most computers don't have a standard high-accuracy, high-precision time clock that is also quick to access.

MSDN Time Functions (HTTP://MSDN.MICROSOFT.COM/LIBRARY/DEFAULT.ASP?URL=/LIBRAI
US/SYSINFO/BASE/TIME_FUNCTIONS.ASP[163])

Timer Function Performance (HTTP://DEVELOPER.NVIDIA.COM/OBJECT/TIMER_-
FUNCTION_PERFORMANCE.HTML[164])

GetTickCount has a precision (dependent on your timer tick rate) of one millisecond, its accuracy typically within a 10-55ms expected error, the best thing is that it increments at a constant rate. (*WaitForSingleObject* uses the same timer).

GetSystemTimeAsFileTime has a precision of 100-nanoseconds, its accuracy is similar to *GetTickCount*.

QueryPerformanceCounter can be slower to obtain but has higher accuracy, uses the HAL (with some help from ACPI) a problem with it is that it can travel back in time on over-clocked PCs due to garbage on the LSBs, note that the functions fail unless the supplied LARGE_INTEGER is DWORD aligned.

162 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPONENT%20OBJECT%20MODEL

163 HTTP://MSDN.MICROSOFT.COM/LIBRARY/DEFAULT.ASP?URL=/LIBRARY/EN-US/SYSINFO/BASE/TIME_FUNCTIONS.ASP

164 HTTP://DEVELOPER.NVIDIA.COM/OBJECT/TIMER_FUNCTION_PERFORMANCE.HTML

Performance counter value may unexpectedly leap forward (HTTP://SUPPORT.MICROSOFT.COM/DEFAULT.ASPX?SCID=KB;EN-US;Q274323&[165])

timeGetTime (via winmm.dll) has a precision of ˜5ms.

File System

MakeSureDirectoryPathExists (via Image Help Library - IMAGHLP.DLL, `#pragma comment( lib, "imagehlp.lib" )`, `#include <imagehlp.h>`) creates directories, only useful to create/force the existence of a given dir tree or multiple directories, or if the linking is already present, note that it is single threaded.

Network

Network applications are often built in C++ on windows utilizing the WinSock API functions.

Resources

Resources files are perhaps one of the most useful elements included on the WIN32 API, they are how we program menu's, add icons, backgrounds, music and many more aesthetically pleasing elements to our programs. Sadly the use in compilation use of resource files is today to those using the MS Visual Studio IDE (resource editor, resource structure understanding).

> **Note:**
> This is similar to the approach of the QT library in dealing with GUI elements, but in this case the OS programing API has direct support for the handling of the resource elements.

The resources are defined in a .rc file (resource c) and are included at the linking phase of compile. Resource files work hand in hand with a header file (usually called resource.h) which carries the definitions of each ID.

For example a simple RC file might contain a menu:

165 HTTP://SUPPORT.MICROSOFT.COM/DEFAULT.ASPX?SCID=KB;EN-US;Q274323&

```
//////////////
IDR_MYMENU MENU
BEGIN
    POPUP "&File"
    BEGIN
         MENUITEM "&About", ID_FILE_ABOUT
        MENUITEM "E&xit", ID_FILE_EXIT
    END
    POPUP "&Edit"
    BEGIN
    // Insert menu here :p
    END

    POPUP "&Links"
    BEGIN
        MENUITEM "&Visit Lukem_95's Website", ID_LINK_WEBSITE
        MENUITEM "G&oogle.com", ID_LINK_GOOGLE
END
END
//////////////
```

And the corresponding H file:

```
#define IDR_MYMENU 9000
#define ID_FILE_EXIT 9001
#define ID_LINK_WEBSITE 9002
#define ID_LINK_GOOGLE 9003 #define ID_FILE_ABOUT 9004
```

166

Win32 API Wrappers

Since the Win32 API is C based and also a moving target and since some alterations are done in each OS version some wrappers were created, in this section you will find some of the approaches available to facilitate the use of the API in a C++ setup and provide abstraction from the low level stuff with higher level imple-

166 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

mentations of common needed features, dealing with the GUI, complex controls even communications and database access.

Microsoft Foundation Classes **(MFC);**

a C++ library for developing Windows applications and UI components. Created by Microsoft for the C++ Windows programmer as an abstraction layer for the Win32 API, the use of the new STL enabled capabilities is scarce on the MFC. It's also compatible with Windows CE (the pocket PC version of the OS). MFC was designed to use the Document-View pattern a variant of the Model View Controller (MVC) pattern.
More info about **MFC** can be obtained on the WINDOWS PROGRAMMING[167] Wikibook.

Windows Template Library **(WTL);**

a C++ library for developing Windows applications and UI components. It extends ATL (Active Template Library) and provides a set of classes for controls, dialogs, frame windows, GDI objects, and more. This library is not supported by Microsoft Services (but is used internally at MS and available for download at MSDN).

Win32 Foundation Classes **(WFC);**

(HTTP://WWW.SAMBLACKBURN.COM/WFC/)[168] a library of C++ classes that extend Microsoft Foundation Classes (MFC) to do NT specific things.

Borland Visual Component Library **(VCL);**

a Delphi/C++ library for developing Windows applications, UI components and different kinds of service applications. Created by Borland as an abstraction layer for the Win32 API, but also implementing many non-visual, and non windows-specific objects, like AnsiString class for example.

Note:
There are more generic wrapper that do not focus exclusively on the Windows API, like the *Qt (framework)* or *WxWidgets* these are covered on the GENERIC WRAPPERS SECTION[a] of the book.

a Chapter 6.6.1 on page 605

167 HTTP://EN.WIKIBOOKS.ORG/WIKI/WINDOWS%20PROGRAMMING%23SECTION_3%3A_MICROSOFT_FOUNDATION_CLASSES_AND_COM

168 HTTP://WWW.SAMBLACKBURN.COM/WFC/)

6.6.2 Generic wrappers

Generic GUI/API wrappers are programming libraries that provide a uniform platform neutral interface (API) to the operating system regardless of underlying platform. Such libraries greatly simplify development of cross-platform software.

Using a wrapper as a portability layer will offer applications some or all following benefits:

- Independence from the hardware.
- Independence from the Operating System.
  - Independence from changes made to specific releases.
  - Independence from API styles and error codes.

Cross-platform programming is more than only GUI programming. Cross-platform programming deals with the minimum requirements for the sections of code that aren't specified by the C++ Standard Language, so as programs can be compiled and run across different hardware platforms.

Here is some cross-platform GUI toolkit:

- **GTKMM**[170] - an interface for the C GUI library GTK+. It is not cross-platform by design, but rather mutli-platform i.e. can be used on many platform.
- **QT**[171] (HTTP://QT.NOKIA.COM)[172] - a cross-platform (Qt is the basis for the Linux KDE desktop environment and supports the X Window System (Unix/X11), Apple Mac OS X, Microsoft Windows NT/9x/2000/XP/Vista/7 and the Symbian OS), it is an object-oriented **application development** framework, widely used for the development of GUI programs (in which case it is known as a widget toolkit), and for developing non-GUI programs such as console tools and servers. Used in numerous commercial applications such as Google Earth, Skype for Linux and Adobe Photoshop Elements. Released under the LGPL or a commercial license.
- **WXWIDGETS**[173] (HTTP://WWW.WXWINDOWS.ORG/)[174] - a widget toolkit for creating graphical user interfaces (GUIs) for cross-platform applications on

169 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
170 HTTP://EN.WIKIPEDIA.ORG/WIKI/GTKMM
171 HTTP://EN.WIKIBOOKS.ORG/WIKI/QT
172 HTTP://QT.NOKIA.COM)
173 HTTP://EN.WIKIPEDIA.ORG/WIKI/WXWIDGETS
174 HTTP://WWW.WXWINDOWS.ORG/)

Win32, Mac OS X, GTK+, X11, Motif, WinCE, and more using one codebase. It can be used from languages such as C++, Python, Perl, and C#/.NET. Unlike other cross-platform toolkits, wxWidgets applications look and feel native. This is because wxWidgets uses the platform's own native controls rather than emulating them. It's also extensive, free, open-source, and mature. wxWidgets is more than a GUI development toolkit it provides classes for files and streams, application settings, multiple threads, interprocess communication, database access and more.

- **FLTK**[175] The "Fast, Light Toolkit"

176

6.6.3 Multi-tasking

MULTI-TASKING[177] is a process by which multiple tasks (also known as PROCESSES[178]), share common processing resources such as a CPU[179].

A computer with a single CPU, will only run one process at a time. By *running* it means that in a specific point in time, the CPU is actively executing instructions for that process. With a single CPU, systems using SCHEDULING[180] can achieve multi-tasking, by which the time of the processor is time-shared by several processes, permitting each to advance their computations, seemingly in parallel. A process runs for some time and another waiting gets a turn.

The act of reassigning a CPU from one task to another one is called a CONTEXT SWITCH[181]. When context switches occur frequently enough, the illusion of PARALLELISM[182] is achieved.

Note:
Context switching has a cost; when deciding to use multi-tasks, a programmer must be aware of trade-offs in performance.

175 HTTP://EN.WIKIPEDIA.ORG/WIKI/FLTK
176 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
177 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPUTER%20MULTITASKING
178 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23PROCESSES
179 HTTP://EN.WIKIPEDIA.ORG/WIKI/CENTRAL%20PROCESSING%20UNIT
180 HTTP://EN.WIKIPEDIA.ORG/WIKI/SCHEDULING%20%28COMPUTING%29
181 HTTP://EN.WIKIPEDIA.ORG/WIKI/CONTEXT%20SWITCH
182 HTTP://EN.WIKIPEDIA.ORG/WIKI/PARALLEL%20COMPUTING

Even on computers with more than one CPU, MULTIPROCESSOR[183] machines, multi-tasking allows many more tasks to be run than there are CPUs.

Operating systems may adopt one of many different SCHEDULING STRATEGIES[184], which generally fall into the following categories:

- In MULTIPROGRAMMING[185] systems, the running task keeps running until it performs an operation that requires waiting for an external event (e.g. reading from a tape) or until the computer's scheduler forcibly swaps the running task out of the CPU. Multiprogramming systems are designed to maximize CPU usage.
- In TIME-SHARING[186] systems, the running task is required to relinquish the CPU, either voluntarily or by an external event such as a HARDWARE INTERRUPT[187]. Time sharing systems are designed to allow several programs to execute apparently simultaneously. The term *time-sharing* used to define this behavior is no longer in use, having been replaced by the term *multi-tasking*.
- In REAL-TIME[188] systems, some waiting tasks are guaranteed to be given the CPU when an external event occurs. Real time systems are designed to control mechanical devices such as industrial robots, which require timely processing.

Multi-tasking has already been successfully integrated into current Operating Systems. Most computers in use today supports running several processes at a time. This is required for systems using SYMMETRIC MULTIPROCESSOR (SMP)[189] in *distributed computing* and MULTI-CORE OR CHIP MULTIPROCESSORS (CMPs)[190] computing, where processors have gone from dual-core to quad-core and core number will continue to increase. Each technology has its specific limitations and applicability, but all these technologies share the common objective of performing concurrent processing.

Note:
Due to the general adoption of the new paradigm it becomes extremely important to prepare your code for it (plan for scalability), understand guarantees regarding parallelization, and select external libraries that provide the required support.

183 HTTP://EN.WIKIPEDIA.ORG/WIKI/MULTIPROCESSOR
184 HTTP://EN.WIKIPEDIA.ORG/WIKI/SCHEDULING%20%28COMPUTING%29
185 HTTP://EN.WIKIPEDIA.ORG/WIKI/MULTIPROGRAMMING
186 HTTP://EN.WIKIPEDIA.ORG/WIKI/TIME-SHARING
187 HTTP://EN.WIKIPEDIA.ORG/WIKI/HARDWARE%20INTERRUPT
188 HTTP://EN.WIKIPEDIA.ORG/WIKI/REAL-TIME%20COMPUTING
189 HTTP://EN.WIKIPEDIA.ORG/WIKI/SYMMETRIC%20MULTIPROCESSING
190 HTTP://EN.WIKIPEDIA.ORG/WIKI/MULTI-CORE%20PROCESSOR

6.6.4 Processes

PROCESS[191]es are independent execution units that contain their own state information, use their own address spaces, and only interact with each other via INTER-PROCESS COMMUNICATION[192] *(IPC)* mechanisms . A process can be said to at least contain one thread of execution (not to be confused to a complete thread construct). Processes are managed by the hosting OS in a process data structure. The maximum number of processes that can run concurrently, depend on the OS and on the available resources of that system.

Child Process

A **child process** (also SPAWN PROCESS[193]), is a process that was created by another process (the PARENT PROCESS[194]), inheriting most of the parent attributes, such as opened files. Each process may create many child processes but will have at most one parent process; if a process does not have a parent this usually indicates that it was created directly by the KERNEL[195].

In UNIX[196], a child process is in fact created (using FORK[197]) as a copy of the parent. The child process can then OVERLAY[198] itself with a different program (using EXEC[199]) as required. The very first process, called INIT[200], is started by the kernel at booting time and never terminates; other parentless processes may be launched to carry out various DAEMON[201] tasks in USERSPACE[202]. Another way for a process to end up without a parent is, if its parent dies leaving an ORPHAN PROCESS[203]; but in this case it will shortly be adopted by init.

191 HTTP://EN.WIKIPEDIA.ORG/WIKI/PROCESS%20%28COMPUTING%29
192 HTTP://EN.WIKIPEDIA.ORG/WIKI/INTER-PROCESS_COMMUNICATION
193 HTTP://EN.WIKIPEDIA.ORG/WIKI/SPAWN%20PROCESS
194 HTTP://EN.WIKIPEDIA.ORG/WIKI/PARENT%20PROCESS
195 HTTP://EN.WIKIPEDIA.ORG/WIKI/KERNEL%20%28COMPUTER%20SCIENCE%29
196 HTTP://EN.WIKIPEDIA.ORG/WIKI/UNIX
197 HTTP://EN.WIKIPEDIA.ORG/WIKI/FORK%20%28OPERATING%20SYSTEM%29
198 HTTP://EN.WIKIPEDIA.ORG/WIKI/OVERLAY%20%28OPERATING%20SYSTEM%29
199 HTTP://EN.WIKIPEDIA.ORG/WIKI/EXEC%20%28OPERATING%20SYSTEM%29
200 HTTP://EN.WIKIPEDIA.ORG/WIKI/INIT
201 HTTP://EN.WIKIPEDIA.ORG/WIKI/DAEMON%20%28COMPUTING%29
202 HTTP://EN.WIKIPEDIA.ORG/WIKI/USERSPACE
203 HTTP://EN.WIKIPEDIA.ORG/WIKI/ORPHAN%20PROCESS

Inter-Process Communication (IPC)

IPC is generally managed by the operating system.

Shared Memory

Most of more recent OSs provide some sort of memory protection. In a Unix system, each process is given its own virtual address space, and the system, in turn, guarantees that no process can access the memory area of another. If an error occurs on a process, only that process memory's contents can be corrupted.

With shared memory, the need of enabling random-access to shared data between different processes is addressed. But declaring a given section of memory as simultaneously accessible by several processes raises the need for control and synchronization, since several processes might try to alter this memory area at the same time.

6.6.5 Multi-Threading

Until recently, the C++ standard did not include any specification or built-in support for multi-threading. Therefore, THREADING[204] had to be implemented using special threading libraries, which are often platform dependent, as an extension to the C++ standard.

Note:
The new C++0x standard supports multi-threading, reducing the need to know multiple APIs and increasing the portability of code.

Some popular C++ threads libraries include:

(This list is not intended to be complete.)

- BOOST[205] - This package includes several libraries, one of which is threads (concurrent programming). the boost threads library is not very full featured, but is complete, portable, robust and in the flavor of the C++ standard. Uses the boost license that is similar to the BSD license.

204 HTTP://EN.WIKIPEDIA.ORG/WIKI/THREAD%20%28COMPUTER%20SCIENCE%29
205 Chapter 6.4.2 on page 588

- INTEL® THREADING BUILDING BLOCKS[206] *(TBB)*[207] offers a rich approach to expressing parallelism in a C++ program. The library helps you take advantage of multi-core processor performance without having to be a threading expert. Threading Building Blocks is not just a threads-replacement library. It represents a higher-level, task-based parallelism that abstracts platform details and threading mechanism for performance and scalability and performance. It is an open source project under the GNU General Public License version two (GPLv2) with the runtime exception.
- ADAPTIVE COMMUNICATION ENVIRONMENT[208] (often referred to as *ACE*) - Another toolkit which includes a portable threads abstraction along with many many other facilities, all rolled into one library. Open source released under a nonstandard but nonrestrictive license.
- ZTHREADS[209] - A portable thread abstraction library. This library is feature rich, deals only with concurrency and is open source licensed under the MIT license.

Of course, you can access the full POSIX and the C language threads interface from C++ and on Windows the API. So why bother with a library on top of that?

The reason is that things like locks are resources that are allocated, and C++ provides abstractions to make managing these things easier. For instance, `boost::scoped_lock<>` uses object construction/destruction to insure that a mutex is unlocked when leaving the lexical scope of the object. Classes like this can be very helpful in preventing deadlock, race conditions, and other problems unique to threaded programs. Also, these libraries enable you to write cross-platform multi-threading code, while using platform-specific function cannot.

In any case when using *threading methodology*, dictates that you must identify **hotspots**, the segments of code that take the most execution time. To determine the best chance at achieving the maximum performance possible, the task can be approached from *bottom-up* and *top-down* to determine those code segments that can run in parallel.

In the *bottom-up* approach, one focus solely on the hotspots in the code. This requires a deep analysis of the call stack of the application to determine the sections of code that can be run in parallel and reduce hotspots. In hotspot sections that

206 HTTP://EN.WIKIPEDIA.ORG/WIKI/INTEL_THREADING_BUILDING_BLOCKS
207 HTTP://WWW.THREADINGBUILDINGBLOCKS.ORG
208 HTTP://EN.WIKIPEDIA.ORG/WIKI/ADAPTIVE%20COMMUNICATION%20ENVIRONMENT
209 HTTP://EN.WIKIPEDIA.ORG/WIKI/ZTHREADS

employ concurrency, it is still required to move that concurrency at a point higher up in the call stack as to increase the GRANULARITY[210] of each thread execution.

Using the *top-down* approach, the focus is on all the parts of the application, in determining what computations can be coded to run in parallel, at a higher level of abstraction. Reducing the level of abstraction until the overall performance gains are sufficient to reach the necessary goals, the benefit being speed of implementation and code re-usability. This is also the best method for archiving a optimal level of GRANULARITY[211] for all computations.

Threads vs. Processes

Both threads and processes are methods of parallelizing an application, its implementation may differ from one OPERATING SYSTEM[212] to another. A process has always one thread of execution, also known as the **primary thread**. In general, a thread is contained inside a process (in the address space of the process) and different threads of the same process share some resources while different processes do not.

Atomicity

Atomicity refers to **atomic** operations that are **indivisible** and/or **uninterruptible**. Even on a single core, you cannot assume that an operation will be atomic. In that regard only when using assembler can one guarantee the atomicity of an operation. Therefore, the C++ standard provides some guarantees as do operating systems and external libraries.

An **atomic operation** can also be seen as any given set of OPERATION[213]s that can be combined so that they appear to the rest of the system to be a single operation with only two possible outcomes: success or failure. This all depends on the level of abstraction and underling guarantees.

All modern processors provide basic **atomic primitives** which are then used to build more complex atomic objects. In addition to atomic read and write operations, most platforms provide an atomic read-and-update operation like TEST-

210 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23COMPUTATION_GRANULARITY
211 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23COMPUTATION_GRANULARITY
212 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPERATING%20SYSTEM
213 HTTP://EN.WIKIPEDIA.ORG/WIKI/INSTRUCTION%20%28COMPUTER%20SCIENCE%29

AND-SET[214] or COMPARE-AND-SWAP[215], or a pair of operations like LOAD-LINK/STORE-CONDITIONAL[216] that only have an effect if they occur atomically (that is, with no intervening, conflicting update). These can be used to implement LOCKS[217], a vital mechanism for multi-threaded programming, allowing invariants and atomicity to be enforced across groups of operations.

Many PROCESSORS[218], especially 32-BIT[219] ones with 64-BIT[220] FLOATING POINT[221] support, provide some read and write operations that are not atomic: one THREAD[222] reading a 64-bit register while another thread is writing to it may see a combination of both "before" and "after" values, a combination that may never actually have been written to the register. Further, only single operations are guaranteed to be atomic; threads arbitrarily performing groups of reads and writes will also observe a mixture of "before" and "after" values. Clearly, invariants cannot be relied on when such effects are possible.

If not dealing with known guaranteed atomic operations, one should rely on the synchronization primitives at the level of abstraction that one is coding to.

Example - One process

For example, imagine a single process is running on a computer incrementing a value in a given MEMORY LOCATION[223]. To increment the value in that memory location:

1. the process reads the value in the memory location;
2. the process adds one to the value;
3. the process writes the new value back into the memory location.

Example - Two processes

214 HTTP://EN.WIKIPEDIA.ORG/WIKI/TEST-AND-SET
215 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPARE-AND-SWAP
216 HTTP://EN.WIKIPEDIA.ORG/WIKI/LOAD-LINK%2FSTORE-CONDITIONAL
217 HTTP://EN.WIKIPEDIA.ORG/WIKI/LOCK%20%28SOFTWARE%20ENGINEERING%29
218 HTTP://EN.WIKIPEDIA.ORG/WIKI/CENTRAL%20PROCESSING%20UNIT
219 HTTP://EN.WIKIPEDIA.ORG/WIKI/32-BIT
220 HTTP://EN.WIKIPEDIA.ORG/WIKI/64-BIT
221 HTTP://EN.WIKIPEDIA.ORG/WIKI/FLOATING%20POINT
222 HTTP://EN.WIKIPEDIA.ORG/WIKI/THREAD%20%28COMPUTER%20SCIENCE%29
223 HTTP://EN.WIKIPEDIA.ORG/WIKI/MEMORY%20LOCATION

Now, imagine two processes are running incrementing a single, shared memory location:

1. the first process reads the value in memory location;
2. the first process adds one to the value;

but before it can write the new value back to the memory location it is suspended, and the second process is allowed to run:

1. the second process reads the value in memory location, the *same* value that the first process read;
2. the second process adds one to the value;
3. the second process writes the new value into the memory location.

The second process is suspended and the first process allowed to run again:

1. the first process writes a now-wrong value into the memory location, unaware that the other process has already updated the value in the memory location.

This is a trivial example. In a real system, the operations can be more complex and the errors introduced extremely subtle. For example, reading a 64-bit value from memory may actually be implemented as two SEQUENTIAL[224] reads of two 32-bit memory locations. If a process has only read the first 32-bits, and before it reads the second 32-bits the value in memory gets changed, it will have neither the original value nor the new value but a mixed-up GARBAGE[225] value.

Furthermore, the specific order in which the processes run can change the results, making such an error difficult to detect and debug.

OS and portability

Considerations are not only necessary with regard to the underling hardware but also in dealing with the different OS APIs. When porting code across different OSs one should consider what guarantees are provided. Similar considerations are necessary when dealing with external libraries.

224 HTTP://EN.WIKIPEDIA.ORG/WIKI/SEQUENCE

225 HTTP://EN.WIKIPEDIA.ORG/WIKI/GARBAGE%20%28COMPUTER%20SCIENCE%29

Note:
For instance on the Macintosh, the set file position call is atomic, whereas on Windows, it's a pair of calls.

Race condition

A **RACE CONDITION**[226] (*data race*, or simply *race*), occurs when data is accessed concurrently from multiple execution paths. It happens for instance when multiple threads have shared access to the same resource such as a file or a block of memory, and at least one of the accesses is a write. This can lead to interference with one another.

Threaded programming is built around predicates and shared data. It is necessary to identify all possible execution paths and identify truly independent computations. To avoid problems it is best to implement concurrency at the highest level possible.

Most *race conditions* occur due to an erroneous assumption about the order in which threads will run. When dealing with shared variables, never assume that a threaded write operation will precede a threaded read operation. If you need guarantees you should see if synchronization primitives are available, and if not, you should implement your own.

Locking

LOCKING[227] temporarily prevents un-shareable resources from being used simultaneously. Locking can be achieved by using a synchronization object.

One of the biggest problems with threading is that locking requires analysis and understanding of the data and code relationships. This complicates software development--especially when targeting multiple operating systems. This makes multi-threaded programming more like art than science.

The number of locks (depending on the synchronization object) may be limited by the OS. A lock can be set to protect more than one resource, if always accessed in the same critical region.

226 HTTP://EN.WIKIPEDIA.ORG/WIKI/RACE_CONDITION%23COMPUTING
227 HTTP://EN.WIKIPEDIA.ORG/WIKI/LOCK_%28COMPUTER_SCIENCE%29

Critical section
A CRITICAL SECTION[228] is a region defined as critical to the parallelization of code execution. The term is used to define code sections that need to be executed in isolation with respect to other code in the program.

This is a common fundamental concept. These sections of code need to be protected by a synchronization technique as they can create *race conditions*.

Deadlock
A DEADLOCK[229] is said to happen whenever there is a lock operation that results in a never-ending waiting cycle among concurrent threads.

Synchronization

Except when used to guarantee the correct execution of a parallel computation, synchronization is an overhead. Attempt to keep it to a minimum by taking advantage of the THREAD'S LOCAL STORAGE[230] or by using exclusive memory locations.

Computation granularity
Computation granularity is loosely defined as the amount of computation performed before any synchronization is needed. The longer the time between synchronizations, the less granularity the computation will have. When dealing with the requirements for parallelism, it will mean being easier to scale to an increased number of threads and having lower overhead costs. A high level of granularity can mean that any benefit from using threads will be lost due to the requirements of synchronization and general thread overhead.

Mutex
MUTEX[231] is an abbreviation for *mutual exclusion*. It relies on a synchronization facility supplied by the operating system (not the CPU). Since this system objects can only be owned by a single thread at any given time, the mutex object facilitates protection against data races and allows for thread-safe synchronization of data

228 HTTP://EN.WIKIPEDIA.ORG/WIKI/CRITICAL_SECTION
229 HTTP://EN.WIKIPEDIA.ORG/WIKI/DEADLOCK
230 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23THREAD%20LOCAL%20STORAGE%20%28TLS%29
231 HTTP://EN.WIKIPEDIA.ORG/WIKI/MUTUAL%20EXCLUSION

between threads. By calling one of the lock functions, the thread obtains ownership of a mutex object, it then relinquishes ownership by calling the corresponding unlock function. Mutexes can be either recursive or non-recursive, and may grant simultaneous ownership to one or many threads.

Semaphore

A SEMAPHORE[232] is a *yielding* synchronization object that can be used to synchronize several threads. This is the most commonly used method for synchronization

Spinlock

SPINLOCKS[233] are *busy-wait* synchronization objects, used as a substitute for Mutexes. They are an implementation of inter-thread locking using machine dependent assembly instructions (such as test-and-set) where a thread simply waits (*spins*) in a loop that repeatedly checks if the lock becomes available (*busy wait*). This is why spinlocks perform better if locked for a short period of time. They are never used on single-CPU machines.

Threads

Threads are by definition a coding construct and part of a PROGRAM[234] that enable it to FORK[235] (or split) itself into two or more simultaneously (or pseudo-simultaneously) running TASK[236]s. Threads use PRE-EMPTIVE MULTITASKING[237].

The thread is the basic unit (the smallest piece of code) to which the operating system can allocate a distinct processor time (schedule) for execution. This means that, threads in reality, don't run concurrently but in sequence on any single core system. Threads often depend on the OS thread scheduler to preempt a busy thread and resume another thread.

The thread today is not only a key concurrency model supported by most if not all modern computers, programming languages, and operating systems but is itself at

232 HTTP://EN.WIKIPEDIA.ORG/WIKI/SEMAPHORE_%28PROGRAMMING%29
233 HTTP://EN.WIKIPEDIA.ORG/WIKI/SPINLOCK
234 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPUTER%20PROGRAM
235 HTTP://EN.WIKIPEDIA.ORG/WIKI/FORK_%28OPERATING_SYSTEM%29
236 HTTP://EN.WIKIPEDIA.ORG/WIKI/TASK%20%28COMPUTERS%29
237 HTTP://EN.WIKIPEDIA.ORG/WIKI/PRE-EMPTIVE%20MULTITASKING

the core of hardware evolution, such as symmetric multi-processors, understanding threads is now a necessity to all programmers.

The order of execution of the threads is controlled by the process scheduler of the OS, it is non-deterministic. The only control available to the programmer is in attributing a priority to the thread but never assume a particular order of execution.

User Interface Thread
This type of distinction is reserved to indicate that the particular thread implements a message map to respond to events and messages generated by user inputs as he interacts with the application. This is especially common when working with the Windows platform (Win32 API) because of the way it implements message pumps.

Worker Thread
This distinction serves to specify threads that do not directly depend or are part of the graphical user interface of the application, and run concurrently with the main execution thread.

Thread local storage (TLS)
The residence of thread local variables, a thread dedicated section of the global MEMORY[238]. Each thread (or fiber) will receive its own stack space, residing in a different memory location. This will consist of both reserved and initially committed memory. That is freed when the thread exits but will not be freed if the thread is terminated by other means.

Since all threads in a PROCESS[239] share the same ADDRESS SPACE[240], it makes data in a static or GLOBAL VARIABLE[241] to be normally located at the same memory location, when referred to by threads from the same process. It is important for software to take in consideration hardware cache coherence. For instance in multiprocessor environments, each processor has a local cache. If threads on different processors modify variables residing on the same cache line, this will invalidate that cache line, forcing a cache update, hurting performance. This is referred to as **false sharing**.

This type of storage is indicated for variables that store temporary or even partial results, since condensing the needed synchronization of the partial results in

238 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPUTER%20STORAGE
239 HTTP://EN.WIKIPEDIA.ORG/WIKI/PROCESS%20%28COMPUTING%29
240 HTTP://EN.WIKIPEDIA.ORG/WIKI/ADDRESS%20SPACE
241 HTTP://EN.WIKIPEDIA.ORG/WIKI/GLOBAL%20VARIABLE

as fewer and infrequent instances possible will contribute to the reduction of synchronization overhead.

Thread Synchronization

The synchronization can be defined in several steps the first is the process lock, where a process is made to halt execution due to find a protected resource locked, there is a cost for locking especially if the lock lasts for too long.

Obviously there is a performance hit if any synchronization mechanism is heavily used. Because they are an expensive operation, in certain cases, increasing the use of TLSs instead of relying only on shared data structures will reduce the need for synchronization.

Critical Section

Suspend and Resume

Synchronizing on Objects

Cooperative vs. Preemptive Threading

Thread pool

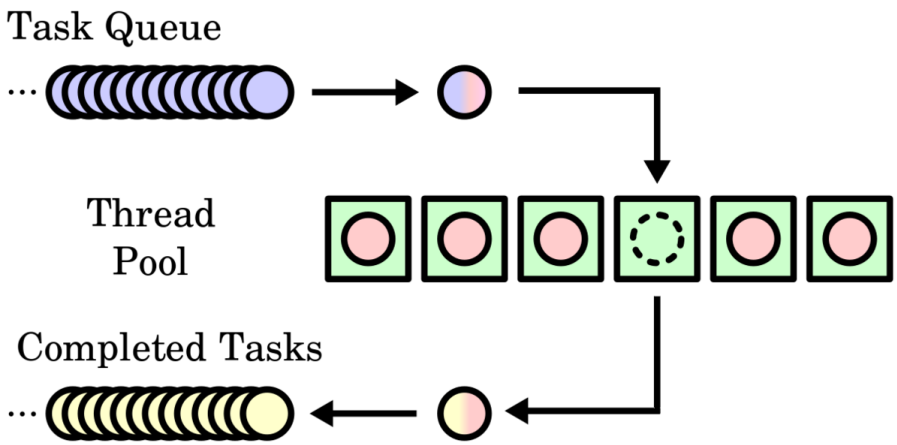


Figure 26: A simple thread pool. The task queue has many waiting tasks (blue circles). When a thread opens up in the queue (green box with dotted circle) a task comes off the queue and the open thread executes it (red circles in green boxes). The completed task then "leaves" the thread pool and joins the completed tasks list (yellow circles)..

Fibers

A FIBER[242] is a particularly lightweight THREAD OF EXECUTION[243]. Like threads, fibers share ADDRESS SPACE[244]. However, fibers use CO-OPERATIVE MULTITASKING[245], fibers yield themselves to run another fiber while executing.

Operating system support

Less support from the OPERATING SYSTEM[246] is needed for fibers than for threads. They can be implemented in modern UNIX[247] systems using the library functions

242 HTTP://EN.WIKIPEDIA.ORG/WIKI/FIBER_%28COMPUTER_SCIENCE%29
243 HTTP://EN.WIKIPEDIA.ORG/WIKI/THREAD%20OF%20EXECUTION
244 HTTP://EN.WIKIPEDIA.ORG/WIKI/ADDRESS%20SPACE
245 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPUTER%20MULTITASKING%23COOPERATIVE%20MULTITASKING%2FTIME-SHARING
246 HTTP://EN.WIKIPEDIA.ORG/WIKI/OPERATING%20SYSTEM
247 HTTP://EN.WIKIPEDIA.ORG/WIKI/UNIX

getcontext, *setcontext* AND *swapcontext*[248] in `ucontext.h`, as in GNU PORTABLE THREADS[249].

On MICROSOFT WINDOWS[250], fibers are created using the *ConvertThreadToFiber* and *CreateFiber* calls; a fiber that is currently suspended may be resumed in any thread. Fiber-local storage, analogous to THREAD-LOCAL STORAGE[251], may be used to create unique copies of variables.

SYMBIAN OS[252] uses a similar concept to fibers in its Active Scheduler. An ACTIVE OBJECT (SYMBIAN OS)[253] contains one fiber to be executed by the Active Scheduler when one of several outstanding asynchronous calls complete. Several Active objects can be waiting to execute (based on priority) and each one must restrict is own execution time.

6.6.6 Exploiting parallelism

Most of the parallel architecture research was done in the 1960s and 1970s, providing solutions for problems that only today are reaching general awareness. As the need of concurrent programming increases, mostly due to today's hardware evolution, we as programmers are pressed to implement programming models that ease the complicated process of dealing with the old thread model in a way it preserves development time by abstracting the problem.

248 HTTP://EN.WIKIPEDIA.ORG/WIKI/SETCONTEXT
249 HTTP://EN.WIKIPEDIA.ORG/WIKI/GNU%20PORTABLE%20THREADS
250 HTTP://EN.WIKIPEDIA.ORG/WIKI/MICROSOFT%20WINDOWS
251 HTTP://EN.WIKIPEDIA.ORG/WIKI/THREAD-LOCAL%20STORAGE
252 HTTP://EN.WIKIPEDIA.ORG/WIKI/SYMBIAN%20OS
253 HTTP://EN.WIKIPEDIA.ORG/WIKI/ACTIVE%20OBJECT%20%28SYMBIAN%20OS%29

OpenMP

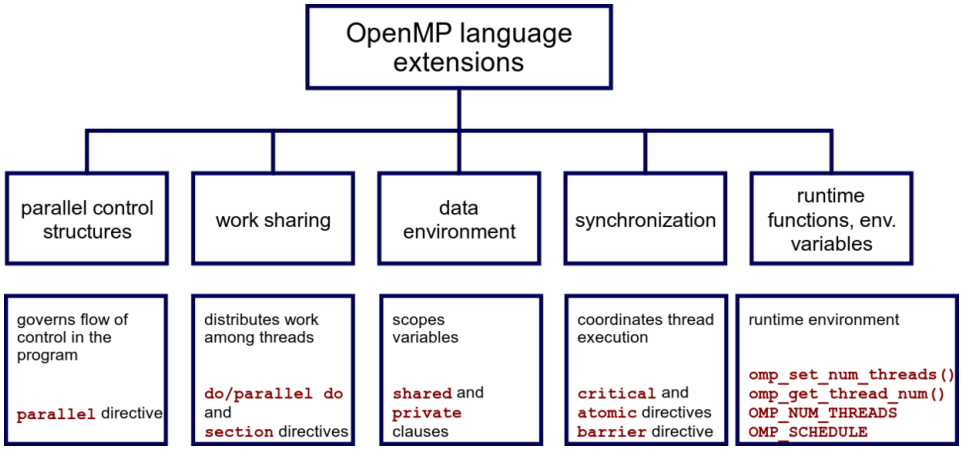


Figure 27: Chart of OpenMP constructs.

6.7 Software Internationalization

INTERNATIONALIZATION AND LOCALIZATION[254] refer to how computer software is adapted for other locations, nations or cultures. This means specifically those that are non-native to the programmer(s) or the primary user group

In specific, internationalization deals with the process of designing a software application in a way that it can be configured or adapted to work with various languages and regions without heavy changes to the code base. On the other hand localization deals with the process of enabling the configuration or auto adaptation of the software to a specific region, timezone or language by adding locale-specific components and text translation.

6.7.1 Text encoding

Text, in particular the characters are used to generate readable text consists on the use of a character encoding scheme that pairs a sequence of characters from a given character set (sometimes referred to as code page) with something else, such as a

254 HTTP://EN.WIKIPEDIA.ORG/WIKI/INTERNATIONALIZATION%20AND%20LOCALIZATION

sequence of natural numbers, octets or electrical pulses, in order to facilitate the use of its digital representation.

A easy to understand example would be Morse code, which encodes letters of the Latin alphabet as series of long and short depressions of a telegraph key; this is similar to how ASCII, encodes letters, numerals, and other symbols, as integers.

Text and data

Probably the most important use for a byte is holding a character code. Characters typed at the keyboard, displayed on the screen, and printed on the printer all have numeric values. To allow it to communicate with the rest of the world, the IBM PC uses a variant of the ASCII character set. There are 128 defined codes in the ASCII CHARACTER SET[255]. IBM uses the remaining 128 possible values for extended character codes including European characters, graphic symbols, Greek letters, and math symbols.

In earlier days of computing, the introduction of coded character sets such as ASCII (1963) and EBCDIC (1964) began the process of standardization. The limitations of such sets soon became apparent, and a number of ad-hoc methods developed to extend them. The need to support multiple writing systems (Languages), including the CJK family of East Asian scripts, required support for a far larger number of characters and demanded a systematic approach to character encoding rather than the previous ad hoc approaches.

6.7.2 What's this about UNICODE?

UNICODE[256] is an industry standard whose goal is to provide the means by which text of all forms and languages can be encoded for use by computers. Unicode 6.1 was released in January 2012 and is the current version. It currently comprises over 109,000 characters from 93 scripts. Since Unicode is just a standard that assigns numbers to characters, there also needs to be methods for encoding these numbers as bytes. The three most common character encodings are UTF-8, UTF-16, and UTF-32, of which UTF-8 is by far the most frequently used.

In the Unicode standard, **PLANES**[257] are groups of numerical values (code points) that point to specific characters. Unicode code points are logically divided into 17

255 Chapter 4.8.1 on page 452
256 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE
257 HTTP://EN.WIKIPEDIA.ORG/WIKI/PLANE%20%28UNICODE%29

planes, each with 65,536 (= 2^{16}) code points. Planes are identified by the numbers 0 to $16_{decimal}$, which corresponds with the possible values 00-$10_{hexadecimal}$ of the first two positions in six position format (*hh***'hhhh***). As of version 6.1, six of these planes have assigned code points (characters), and are named.*

Plane 0 - Basic Multilingual Plane (BMP)

Plane 1 - Supplementary Multilingual Plane (SMP)

Plane 2 - Supplementary Ideographic Plane (SIP)

Planes 3–13 - Unassigned

Plane 14 - Supplement-ary Special-purpose Plane (SSP)

Planes 15–16 - Supplement-ary Private Use Area (S PUA A/B)

BMP and SMP

BMP		**SMP**	
0000–0FFF[258]	8000–8FFF[259]	10000–10FFF[260]	18000-18FFF
1000–1FFF[261]	9000–9FFF[262]	11000–11FFF[263]	19000-19FFF

258 http://en.wikibooks.org/wiki/Unicode%2FCharacter%20reference%2F0000-0FFF
259 http://en.wikibooks.org/wiki/Unicode%2FCharacter%20reference%2F8000-8FFF
260 http://en.wikibooks.org/wiki/Unicode%2FCharacter%20reference%2F10000-10FFF
261 http://en.wikibooks.org/wiki/Unicode%2FCharacter%20reference%2F1000-1FFF
262 http://en.wikibooks.org/wiki/Unicode%2FCharacter%20reference%2F9000-9FFF
263 http://en.wikibooks.org/wiki/Unicode%2FCharacter%20reference%2F11000-11FFF

2000–2FFF[264]	A000–AFFF[265]	12000–12FFF[266]	1A000-1AFFF
3000–3FFF[267]	B000–BFFF[268]	13000–13FFF[269]	1B000-1BFFF[270]
4000–4FFF[271]	C000–CFFF[272]	14000-14FFF	1C000-1CFFF
5000–5FFF[273]	D000–DFFF[274]	15000-15FFF;	1D000–1DFFF[275]
6000–6FFF[276]	E000–EFFF[277]	16000–16FFF[278]	1E000-1EFFF
7000–7FFF[279]	F000–FFFF[280]	17000-17FFF	1F000–1FFFF[281]

ISP and SSP

264 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F2000-2FFF

265 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FA000-AFFF

266 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F12000-12FFF

267 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F3000-3FFF

268 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FB000-BFFF

269 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F13000-13FFF

270 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F1B000-1BFFF

271 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F4000-4FFF

272 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FC000-CFFF

273 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F5000-5FFF

274 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FD000-DFFF

275 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F1D000-1DFFF

276 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F6000-6FFF

277 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FE000-EFFF

278 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F16000-16FFF

279 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F7000-7FFF

280 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FF000-FFFF

281 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F1F000-1FFFF

SIP		SSP
20000–20FFF[282]	28000–28FFF[283]	E0000–E0FFF[284]
21000–21FFF[285]	29000–29FFF[286]	
22000–22FFF[287]	2A000–2AFFF[288]	
23000–23FFF[289]	2B000–2BFFF[290]	1BFFF;
24000–24FFF[291]		
25000–25FFF[292]		
26000–26FFF[293]		
27000–27FFF[294]	2F000–2FFFF[295]	

PUA

282 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F20000-20FFF
283 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F28000-28FFF
284 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FE0000-E0FFF
285 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F21000-21FFF
286 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F29000-29FFF
287 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F22000-22FFF
288 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F2A000-2AFFF
289 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F23000-23FFF
290 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F2B000-2BFFF
291 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F24000-24FFF
292 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F25000-25FFF
293 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F26000-26FFF
294 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F27000-27FFF
295 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F2F000-2FFFF

PUA

PUA

F0000–F0FFF[296]	F8000–F8FFF[297]	100000–100FFF[298]	108000–108FFF[299]
F1000–F1FFF[300]	F9000–F9FFF[301]	101000–101FFF[302]	109000–109FFF[303]
F2000–F2FFF[304]	FA000–FAFFF[305]	102000–102FFF[306]	10A000–10AFFF[307]
F3000–F3FFF[308]	FB000–FBFFF[309]	103000–103FFF[310]	10B000–10BFFF[311]
F4000–F4FFF[312]	FC000–FCFFF[313]	104000–104FFF[314]	10C000–10CFFF[315]

296 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FF0000-F0FFF

297 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FF8000-F8FFF

298 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F100000-100FFF

299 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F108000-108FFF

300 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FF1000-F1FFF

301 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FF9000-F9FFF

302 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F101000-101FFF

303 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F109000-109FFF

304 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FF2000-F2FFF

305 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FFA000-FAFFF

306 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F102000-102FFF

307 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F10A000-10AFFF

308 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FF3000-F3FFF

309 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FFB000-FBFFF

310 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F103000-103FFF

311 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F10B000-10BFFF

312 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FF4000-F4FFF

313 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FFC000-FCFFF

314 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F104000-104FFF

315 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F10C000-10CFFF

PUA			
F5000–F5FFF[316]	FD000–FDFFF[317]	105000–105FFF[318]	10D000–10DFFF[319]
F6000–F6FFF[320]	FE000–FEFFF[321]	106000–106FFF[322]	10E000–10EFFF[323]
F7000–F7FFF[324]	FF000–FFFFF[325]	107000–107FFF[326]	10F000–10FFFF[327]

Currently, about ten percent of the potential space is used. Furthermore, ranges of characters have been tentatively mapped out for every current and ancient writing system (script) the Unicode consortium has been able to identify. While Unicode may eventually need to use another of the spare 11 planes for ideographic characters, other planes remain. Even if previously unknown scripts with tens of thousands of characters are discovered, the limit of 1,114,112 code points is unlikely to be reached in the near future. The Unicode consortium has stated that limit will never be changed.

The odd-looking limit (it is not a power of 2), is not due to UTF-8, which was designed with a limit of 2^{31} code points (32768 planes), and can encode 2^{21} code points (32 planes) even if limited to 4 bytes but is due to the design of UTF-16. In UTF-16 a "surrogate pair" of two 16-bit WORDS[328] is used to encode 2^{20} code points 1 to 16, in addition to the use of single words to encode plane 0.

316 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FF5000-F5FFF
317 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FFD000-FDFFF
318 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F105000-105FFF
319 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F10D000-10DFFF
320 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FF6000-F6FFF
321 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FFE000-FEFFF
322 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F106000-106FFF
323 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F10E000-10EFFF
324 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FF7000-F7FFF
325 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2FFF000-FFFFF
326 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F107000-107FFF
327 HTTP://EN.WIKIBOOKS.ORG/WIKI/UNICODE%2FCHARACTER%20REFERENCE%2F10F000-10FFFF
328 Chapter 3.3.1 on page 122

UTF-8

UTF-8[329] is a variable-length encoding of Unicode, using from 1 to 4 bytes for each character. It was designed for compatibility with ASCII, and as such, single-byte values represent the same character in UTF-8 as they do in ASCII. Because a UTF-8 stream doesn't contain '\0's, you may use it directly in your existing C++ code without any porting (except when counting the 'actual' number of character in it).

UTF-16

UTF-16[330] is also variable-length, but works in 16 bit units instead of 8, so each character is represented by either 2 or 4 bytes. This means that it is not compatible with ASCII.

UTF-32

Unlike the previous two encodings, UTF-32 is not variable-length: every character is represented by exactly 32-bits. This makes encoding and decoding easier, because the 4-byte value maps directly to the Unicode code space. The disadvantage is in space efficiency, as each character takes 4 bytes, no matter what it is.

331

6.8 Optimizations

Optimization can be regarded as a directed effort to increase the performance of something, an important concept in engineering, in particular, the case of Software engineering that we are covering. We will deal with specific computational tasks and best practices to reduce resources utilizations, not only of system resources but also of programmers and users, all based in optimal solutions evolved from the empirical validating of hypothesis and logical steps.

All optimization steps taken should have as a goal the reduction of requirements and the promotion of the program objectives. Any claims can only be substantiated

329 HTTP://EN.WIKIPEDIA.ORG/WIKI/UTF-8
330 HTTP://EN.WIKIPEDIA.ORG/WIKI/UTF-16
331 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

by PROFILING[332] the given problem and the applied solution. **Without profiling any optimization is moot**.

Optimization is often a topic of discussion among programmers and not all conclusions may be consensual, since they are very closely related to the goals, the programmer experience, and dependent of specific setups. The level of optimization will mostly depend directly from actions and decisions the programmer makes. Those can be simple things, from basic coding practices to the selection of the tools one choses to use to create the program. Even selecting the right compiler will have an impact. A good optimizing compiler permits the programmer to define his aspirations for the optimized outcome; how good the compiler is at optimizing depends on the level of satisfaction the programmer gets from the resulting compilation.

6.8.1 Code

One of the safest ways of optimization is to reduce complexity, ease organization and structure and at the same time evading code bloat. This requires the capacity to plan without losing track of future needs, in fact it is a compromise the programmer makes between a multitude of factors.

Code optimization techniques, fall into the categories of:

- High Level Optimization
  - Algorithmic Optimization (Mathematical Analysis)
  - Simplification
- Low Level Optimization
  - Loop Unrolling
  - Strength Reduction
  - Duff's Device
  - Clean Loops

KISS

The "keep it simple, stupid" (KISS[333]) principle, calls for giving simplicity a high priority in development. It is very similar to a maxim from Albert Einstein's that states, "everything should be made as simple as possible, but no simpler.", the difficulty for many adopters have is to determine what level of simplicity should

332 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23PROFILING
333 HTTP://EN.WIKIPEDIA.ORG/WIKI/%3AKISS%20PRINCIPLE

be maintained. In any case, analysis of basic and simpler system is always easier, removing complexity will also open the door for code reutilization and a more generic approach to tasks and problems.

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."
—Martin Golding

Code cleanup

Most of the benefits of a code cleanup should be evident to the experienced programmer, they become a second nature due to the adoption of good programming style guidelines. But as in any human activity, errors will occur and exceptions made, so, in this section we will try to remember the small changes that can have an impact on the optimization of your code.

the use of virtual member functions

Remember the cost on performance of virtual members functions (covered when introducing the VIRTUAL KEYWORD[334]). At the time optimization becomes an issue most project design change regarding optimization will not be possible, but artifacts may remain to be cleaned up. Guaranteeing that no superfluous use of virtual (like in the leaf nodes of your class/structure inheritance trees), will permit other optimizations to occur (i.e.: compiler OPTIMIZED INLINE[335]).

The right data in the right container

One of the top bottleneck on today's systems is dealing with memory CACHES[336], be it CPU CACHE[337] or the physical memory resources, even if PAGING[338] problems are becoming increasingly rare. Since the data (and the load level) a program will handle is highly predictable at the design level, the better optimizations still fall to the programmer.

334 Chapter 4.3.1 on page 394
335 HTTP://EN.WIKIBOOKS.ORG/WIKI/%23AUTO%20INLINE
336 HTTP://EN.WIKIPEDIA.ORG/WIKI/CACHE
337 HTTP://EN.WIKIPEDIA.ORG/WIKI/CPU%20CACHE
338 HTTP://EN.WIKIPEDIA.ORG/WIKI/PAGING

One should store the appropriate data structure in the appropriate container, prefer storing pointers to objects rather than the objects themselves, use "smart" pointers (see the Boost library) and don't attempt to store auto_ptr<> in STL containers, it is not allowed by the Standard, but some implementations are known to incorrectly allow it.

Avoid removing and inserting elements in the middle of a container, doing it at the end of the container has less overhead. Use STL containers when the number of objects is unknown; use static array or buffer when it is known. This requires the understanding of not only each container, but its O(x) guarantees.

Take as an example the STL containers on the issue of using (`myContainer.empty()`) versus (`myContainer.size() == 0`), it is important to understand that depending on the container type or its implementation, the size member function might have to count the number of objects before comparing it to zero. This is very common with list type containers.

While the STL attempts to provides optimal solutions to general cases, if performance does not match your requirements think about writing your own optimal solution for your case, maybe a custom container (probably based on vector) that does not call individual object destructors and uses custom allocators that avoid the delete time overhead.

Using pre-allocation of memory can provide some speed gains and be as simple remember to use the STL vector<T>::reserve() if permitted. Optimize the use system's memory and the target hardware. In today's systems, with virtual memory, threads and multiple-cores (each with its own cache) where I/O operations on the main memory and the amount of time spent moving it around, can slow things down. This can become a performance bottleneck. Instead opt for array-based data structures (cache-coherent data structures), like the STL vector, because data is stored contiguously in memory, over pointer-linked data structures as in linked lists. This will avoid "death by swapping", as the program needs to access highly fragmented data, and will even help the memory pre-fetch that most modern processors do today.

Whenever possible avoid returning containers by value, pass containers by reference.

Consider security costs

Security always has a cost, even in programming. For any algorithm, adding checks, will result in increase the number of steps it takes to finish. As languages

get more complex and abstract, understanding all the finer details (and remember them) increases the time needed to obtain the required experience. Sadly most of the steps taken by some of the implementors of the C++ language lack visibility to the programmer and since they are outside of the standard language, aren't often learned. Remember to familiarized yourself with any extensions or particularities of the C++ implementation you are using.

As a language that puts the power of decision into the programmer's hands, C++ provides several instances where the a similar result can be archived by similar but distinct means. Understanding the sometimes subtle differences is important. For instance, when deciding the needed requirements in ACCESSING MEMBERS OF A STD::VECTOR[339], you can chose [], at() and the an iterator, all have similar results but with distinct performance costs and security considerations.

Code reutilization

Optimization is also reflected on the effectiveness of a code. If you can use an already existing code base/framework that a considerable number of programmers have access to, you can expect it to be less buggy and optimized to solve your particular need.

Some of these code repositories are available to programmers as libraries. Be careful to consider dependencies and check how implementation is done: if used without considerations this can also lead to code bloat and increased memory footprint, as well as decrease the portability of the code. We will take a close look at them in the LIBRARIES SECTION[340] of the book.

To increase code reutilization you will probably fragment the code in smaller sections, files or code, remember to equate that more files and overall complexity also increases compile time.

Function and algorithmic optimizations

When creating a function or a algorithm to address a specific problem sometimes we are dealing with mathematical structures that are specifically indicated to be

339 Chapter 5.2.2 on page 503
340 Chapter 6.3.3 on page 584

optimized by established methods of mathematical minimization, this falls into the specific field of ENGINEERING ANALYSIS FOR OPTIMIZATION[341].

Use of `inline`

As seen before when examining the `inline` keyword, it allows the definition of an inline type of function, that works similarly to LOOP UNWINDING[342] for increasing code performance. A non-inline function requires a call instruction, several instructions to create a stack frame, and then several more instructions to destroy the stack frame and return from the function. By copying the body of the function instead of making a call, the size of the machine code increases, but the execution time *decreases*.

In addition to using the `inline` keyword to declare an inline function, optimizing compilers may decide to make other functions inline as well (see COMPILER OPTIMIZATIONS[343] section).

ASM

If portability is not a problem and you are proficient with assembler you can use it to optimize computational bottlenecks, even looking at the output of a disassembler will often help looking for ways to improve it. Using ASM in your code brings to the table some other problems (maintainability for instance) so use it at a last resort in you optimization process, and if you use it be sure to document what you have done well.

The X86 DISASSEMBLY[344] Wikibook provides some OPTIMIZATION EXAMPLES[345] using x86 ASM code.

Note:
If using the gcc compiler, the -S option will output the compilation generated assembly.

341 HTTP://EN.WIKIBOOKS.ORG/WIKI/ENGINEERING%20ANALYSIS%2FOPTIMIZATION
342 HTTP://EN.WIKIBOOKS.ORG/WIKI/X86%20DISASSEMBLY%2FCODE%20OPTIMIZATION%23LOOP_UNWINDING
343 Chapter 6.8.3 on page 634
344 HTTP://EN.WIKIBOOKS.ORG/WIKI/X86%20DISASSEMBLY
345 HTTP://EN.WIKIBOOKS.ORG/WIKI/X86%20DISASSEMBLY%2FOPTIMIZATION%20EXAMPLES

6.8.2 Reduction of compile time

Some projects may take a long time to compile. To reduce the time it takes to finish compiling the first step is to check if you have the any Hardware deficiencies. You may be low in resources like memory or just have a slow CPU, even having your HD with a high level of fragmentation can increase compile time.

On the other side, problems may not be due to hardware limitations but in the tools you use, check if you are using the right tools for the job at hand, see if you have the latest version, or if do, if that is what is causing trouble, some incompatibilities may result from updates. In compilers new is always better, but you should check first what has changed and if it serves your purposes.

Experience tells that most likely if you are suffering from slow compile times, the program you are trying to compile was probably poorly designed, check the structure of object dependencies, the includes and take some the time to structure your own code to minimize re-compilation after changes if the compile time justifies it.

Use pre-compiled headers and external header guards this will reduce the work done by the compiler.

6.8.3 Compiler optimizations

COMPILER OPTIMIZATION[346] is the process of tuning, mostly automatically, the output of a compiler in an attempt to improve the operations the programmer has requested, so to minimize or maximize some attribute of an compiled program **while ensuring the result is identical**. By rilling in the compiler optimization programmers can write more intuitive code, and still have them execute in a reasonably fast way, for instance skipping the use of PRE-INCREMENT/DECREMENT OPERATORS[347].

Generally speaking, optimizations are not, and can not be, defined on the C++ standard. The standard sets rules and best practices that dictate a normalization of inputs and outputs. The C++ standard itself permits some latitude on how compilers perform their task since some sections are marked as implementation dependent but generally a base line is established, even so some vendors/implementors do creep in some singular characteristic apparently for security and optimization benefits.

346 HTTP://EN.WIKIPEDIA.ORG/WIKI/COMPILER%20OPTIMIZATION
347 Chapter 3.4.3 on page 166

One notion that is good to keep in mind is that there is not a perfect C++ compiler, but most recent compilers will do several simple optimizations by default, that attempt to abstract and take advantage of existing deeper hardware optimizations or specific characteristics of the target platform, most of these optimizations are almost always welcomed but it is up to the programmer still to have and idea of what is going on and if indeed they are beneficial. As a result it is highly recommended to examine your compiler documentation on how it operates and what optimizations are under the programmer's control, just because a compiler can make some optimization in theory does not mean that it will or even that it will result in an optimization.

The most common compiler optimizations options available to the programmer fall into three categories:

- **Speed**; improving the runtime performance of the generated object code. This is the most common optimization
- **Space**; reducing the size of the generated object code
- **Safety**; reducing the possibility of data structures becoming corrupted (for example, ensuring that an illegal array element is not written to)

Unfortunately, many "speed" optimizations make the code larger, and many "space" optimizations make the code slower -- this is known as the SPACE-TIME TRADEOFF[348].

auto-`inline`

Auto-inlining is similar to implicit inline. Inlining can be an optimization, or a pessimization depending on the code and optimization options selected.

Making use of extended instructions sets

GPU

6.8.4 Run time

As we have seen before runtime is the duration of a program execution, from beginning to termination. This is were all resources needed to run the compiled code

348 HTTP://EN.WIKIPEDIA.ORG/WIKI/SPACE-TIME%20TRADEOFF

are allocated and hopefully released, this is the final objective of any program to be executed, as such it should be the target for ultimate optimizations.

6.8.5 Memory footprint

In the past computer memory has been expensive and technologically limited in size, and scarce resource for programmers. Large amounts of ingenuity was spent in implement complex programs and process huge amounts of data using as little as possible of this resource. Today, modern systems contain enough memory for most usages but capacity demands and expectations have increased as well; as such, techniques to minimize memory usage may still be essential and in fact operational performance has gained a new momentum with the increasing importance of mobile computing.

Measuring the memory usage of a program is difficult and time consuming, and the more complex the program is the harder it becomes to get good metrics. One other side of the problem is that there are no standard benchmarks (not all memory use is equal) or practices to deal with the problem beyond the most basic and generic considerations.

Note:
Take in consideration that performing memory tests in a debug compile will not, in most circumstances, produce any valid insight on memory use, at best it can provide you an indication of the expected ceiling for memory use in the tested functions.

Remember to use `swap()` on `std::vector` (or deque).

When attempting to reduce reduce (or zero) the size of a vector or deque using the `swap()`, on a standard container of that type, will guarantee that the memory is released and no overhead buffer for growth is used. It will also avoid the fallacy of using `erase()` or `reserve()` that will not reduce the memory footprint.

Lazy initialization

It is always needed to maintain the balance between the performance of the system and the resource consumption. Lazy instantiation is one memory conservation mechanism, by which the object initialization is deferred until it is required.

Look at the following example:

```
#include <iostream>

class Wheel {
        int speed;
    public:
        int getSpeed(){
            return speed;
        }
        void setSpeed(int speed){
            this->speed = speed;
        }
};

class Car{
    private:
        Wheel wheel;
    public:
        int getCarSpeed(){
            return wheel.getSpeed();
        }
        char *getName(){
            return "My Car is a Super fast car";
        }
};

int main(){
    Car myCar;
    std::cout << myCar.getName();
}
```

Instantiation of class Car by default instantiates the class Wheel. The purpose of the whole class is to just print the name of the car. Since the instance wheel doesn't serve any purpose, initializing it is a complete resource waste.

It is better to defer the instantiation of the un-required class until it is needed. Modify the above class Car as follows:

```
class Car{
    private:
        Wheel *wheel;
    public:
        Car() {
            wheel=NULL; // a better place would be in the class constructor
 initialization list
        }
        ~Car() {
             if (wheel) {
                 delete wheel;
             }
        }
        int getCarSpeed(){
            if(wheel == NULL){
                wheel = new Wheel();
```

```
            }
            return wheel->getSpeed();
        }
        char *getName(){
            return "My Car is a Super fast car";
        }
};
```

Now the Wheel will be instantiated only when the member function getCarSpeed() is called.

6.8.6 Parallelization

As seen when examining THREADS[349], they can be a "simple" form of taking advantage of hardware resources and optimize the speed performance of a program. When dealing with thread you should remember that it has a cost in complexity, memory and if done wrong when synchronization is required it can even reduce the speed performance, if the design permits it is best to allow threads to run as unencumbered as possible.

6.8.7 I/O reads and writes

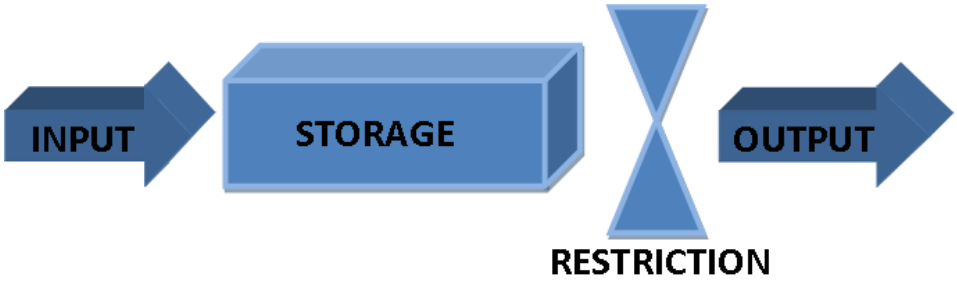


Figure 28: A Schematic of a Queue System

349 Chapter 6.6.2 on page 606

6.8.8 Profiling

Profiling is a form of DYNAMIC PROGRAM ANALYSIS[350] (as opposed to STATIC CODE ANALYSIS[351]), consists in the study of program's behavior using information gathered as the program executes. its purpose is usually to determine which sections of a program to optimize. Mostly by determining which parts of a program are taking most of the execution time, causing bottleneck on accessing resources or the level of access to those resources.

Global clock execution time should be the bottom line when comparing applications performance. Select your algorithms by examining the asymptotic order of executions, as in a parallel setup they will continue to give the best performance. In the case you find an hotspot that can not be parallelized, even after examining higher levels on the call stack, then you should attempt to find a slower but parallelizable algorithm.

branch-prediction profiler

call-graph generating cache profiler

line-by-line profiling

heap profiler

Profiler

Free Profiling tools

- Valgrind (HTTP://VALGRIND.ORG/[352]) an instrumentation framework for building dynamic analysis tools. Includes a cache and branch-prediction profiler, a call-graph generating cache profiler, and a heap profiler. It runs on the

350 HTTP://EN.WIKIPEDIA.ORG/WIKI/DYNAMIC%20PROGRAM%20ANALYSIS
351 HTTP://EN.WIKIPEDIA.ORG/WIKI/STATIC%20CODE%20ANALYSIS
352 HTTP://VALGRIND.ORG/

following platforms: X86/Linux, AMD64/Linux, PPC32/Linux, PPC64/Linux, and X86/Darwin (Mac OS X). Open Source under the GNU General Public License, version 2.
- GNU gprof (HTTP://WWW.GNU.ORG/SOFTWARE/BINUTILS/[353]) a profiler tool. The program was first was introduced on the SIGPLAN Symposium on Compiler Construction in 1982, and is now part of the binutils that are available in mostly all flavors of UNIX. It is capable of monitoring time spent in functions (or even source code lines) and calls to/from them. Open Source under the GNU General Public License.

6.9 Further reading

- OPTIMIZING C++[354]

[355] W:UNIFIED MODELING LANGUAGE[356]

6.10 Modeling Tools

Long gone are the days when you had to do all software designing planing with pencil and paper, it's known that bad design can impact the quality and maintainability of products, affecting time to market and long term profitability of a project.

The solution seems to be CASE and modeling tools which improve the design quality and help to implement design patterns with ease that in turn help to improve design quality, auto documentation and the shortening the development life cycles.

6.10.1 UML (Unified Modeling Language)

Since the late 80s and early 90s, the software engineering industry as a whole was in need of standardization, with the emergence and proliferation of many new competing software design methodologies, concepts, notations, terminologies, processes, and cultures associated with them, the need for unification was self evident by the sheer number of parallel developments. A need for a common

353 HTTP://WWW.GNU.ORG/SOFTWARE/BINUTILS/
354 HTTP://EN.WIKIBOOKS.ORG/WIKI/OPTIMIZING%20C%2B%2B
355 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
356 HTTP://EN.WIKIPEDIA.ORG/WIKI/UNIFIED%20MODELING%20LANGUAGE

ground on the representation of software design was badly needed and to archive it a standardization of geometrical figures, colors, and descriptions.

The UML (Unified Modeling Language) was specifically created to serve this purpose and integrates the concepts of BOOCH[357] (Grady Booch is one of the original developers of UML and is recognized for his innovative work on software architecture, modeling, and software engineering processes), OMT[358], OOSE[359], CLASS-RELATION[360] and OORAM[361]and by fusing them into a single, common and widely usable modeling language tried to be the unifying force, introducing a standard notation that was designed to transcend programming languages, operating systems, application domains and the needed underlying semantics with which programmers could describe and communicate. With its adoption in November 1997 by the OMG (OBJECT MANAGEMENT GROUP[362]) and its support it has become an industry standard. Since then OMG has called for information on object-oriented methodologies, that might create a rigorous software modeling language. Many industry leaders had responded in earnest to help create the standard, the last version of UML (v2.0) was released in 2004.

UML is still widely used by the software industry and engineering community. In later days a new awareness has emerged (commonly called UML fever) that UML *per se* has limitations and is not a good tool for all jobs. Careful study on how and why it is used is needed to make it useful.

363

6.11 Chapter Summary

1. RESOURCE ACQUISITION IS INITIALIZATION (RAII)[364]
2. GARBAGE COLLECTION (GC)[365]

357 HTTP://EN.WIKIPEDIA.ORG/WIKI/GRADY%20BOOCH
358 HTTP://EN.WIKIPEDIA.ORG/WIKI/OBJECT-MODELING%20TECHNIQUE
359 HTTP://EN.WIKIPEDIA.ORG/WIKI/OBJECT-ORIENTED%20SOFTWARE%20ENGINEERING
360 HTTP://EN.WIKIPEDIA.ORG/WIKI/CLASS-RELATION
361 HTTP://EN.WIKIPEDIA.ORG/WIKI/OBJECT%20ORIENTED%20ROLE%20ANALYSIS%20METHOD
362 HTTP://EN.WIKIPEDIA.ORG/WIKI/OBJECT%20MANAGEMENT%20GROUP
363 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
364 Chapter 6 on page 537
365 Chapter 6.1 on page 540

3. Design patterns[366] - Creational[367], Structural[368] and Behavioral[369] patterns.
4. Libraries[370] - APIs vs Frameworks[371] and Static and dynamic libraries[372].
5. Boost library[373]
6. Optimizing your programs[374]
7. Cross-platform development[375]
   a) Win32 (aka WinAPI)[376] - including Win32 wrappers[377].
   b) Cross-platform wrappers[378]
   c) Multitasking[379]
8. Software internationalization[380]
   a) Text encoding[381]
9. Unified Modeling Language (UML)[382]

5[383]

5[384]

366 Chapter 6.2 on page 541
367 Chapter 6.3 on page 543
368 Chapter 6.3.1 on page 559
369 Chapter 6.3.2 on page 564
370 Chapter 6.3.3 on page 584
371 Chapter 6.4 on page 585
372 Chapter 6.4.1 on page 586
373 Chapter 6.4.2 on page 588
374 Chapter 6.7.2 on page 628
375 Chapter 6.5.6 on page 597
376 Chapter 6.6 on page 598
377 Chapter 6.6.1 on page 603
378 Chapter 6.6.1 on page 605
379 Chapter 6.6.2 on page 606
380 Chapter 6.6.6 on page 621
381 Chapter 6.7 on page 621
382 Chapter 6.9 on page 640
383 http://en.wikibooks.org/wiki/Category%3AC%2B%2B%20Programming
384 http://en.wikibooks.org/wiki/Category%3AC%2B%2B%20Programming

7 Appendix A: Internal References

- List of Keywords

[included next to The Compiler]

- List of Standard Headers

[included in The preprocessor Chapter (next to the #include keyword)]

- Table of Preprocessors

[included in The preprocessor Chapter]

- Table of Operators

[included in the introduction to Operators]

- Table of Data Types

[included in the introduction to Variables]

To prevent duplication of content references are removed you may find them on the given locations.

8 Appendix B: External References

8.1 Reference Sites

- http://www.research.att.com/~bs/C++.html

Bjarne Stroustrup's C++ page.

- HTTP://WWW.OPEN-STD.ORG/JTC1/SC22/WG21/DOCS/LIBRARY_-TECHNICAL_REPORT.HTML[1]

C++ Standard Library Technical Report.

- http://www.open-std.org/jtc1/sc22/wg21/

C++ Standards Committee's official website, previously at HTTP://ANUBIS.DKUUG.DK/JTC1/SC22/WG21/[2] , ISO/IEC JTC1/SC22/WG21 is the international standardization working group for the programming language C++.

- http://www.sgi.com/tech/stl/index.html

The SGI Standard Template Library Programmer's Guide.

1 HTTP://WWW.OPEN-STD.ORG/JTC1/SC22/WG21/DOCS/LIBRARY_TECHNICAL_REPORT.HTML

2 HTTP://ANUBIS.DKUUG.DK/JTC1/SC22/WG21/

8.2 Compilers and IDEs

8.2.1 Free or with free versions

- http://gcc.gnu.org/

GCC , the GNU Compiler Collection, which includes a compiler for C++.

- http://www.mingw.org/

MinGW, a Win32 port of the GNU Compiler Collection and toolset designed for compatibility with the host OS.

- http://sourceware.cygnus.com/cygwin

Cygwin, a Win32 port of GCC and GNU Utils designed to simulate a Unix-style environment.

- http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express

The *Microsoft Visual C++ 2010 Express Edition*. It also allows you to build applications that target the Common Language Runtime (CLR). You should read their license for yourself to make sure. MFC, ATL and the Windows headers/libraries are not included with this version. To create Windows programs, you will need to DOWNLOAD THE MICROSOFT PLATFORM SDK[3] as well (for the Windows headers and import libraries).

- http://hpgcc.sourceforge.net/

HP-GCC comprises the GNU C compiler targeted at the ARM processor of ARM-based HP calculators (like the HP49g+), HP specific libraries, a tool (ELF2HP) that converts the gcc produced binary to the appropriate format for the HP calculator, and an emulator (ARM Toolbox/ARM Launcher) that lets you

3 HTTP://WWW.MICROSOFT.COM/DOWNLOADS/DLX/EN-US/LISTDETAILSVIEW.ASPX?FAMILYID=6B6C21D2-2006-4AFA-9702-529FA782D63B

execute ARM programs on your computer. At present, only a Windows version is available, but the site says that Linux and Mac OS X versions are "on the way".

- http://www.ultimatepp.org/

Ultimate++ a C++ cross-platform and Open Source rapid application development suite focused on programmers productivity. It includes a set of libraries (GUI, SQL, etc..), and an integrated development environment.
The IDE can work with GCC, MinGW and Visual C++ 7.1 or 8.0 compilers (including free Visual C++ Toolkit 2003 and Visual C++ 2005 Express Edition) and contains a full featured debugger.

- http://www.codelite.org/

CodeLite, open-source under the terms of the GPL license, cross platform IDE for the C/C++ programming languages (tested on Windows XP SP3, (K)Ubuntu 8.04, and Mac OSX 10.5.2).

- http://www.codeblocks.org/

Code::Blocks, C++ cross-platform and Open Source (GPL2) IDE, runs on Linux or Windows (uses wxWidgets), supports GCC (MingW/Linux GCC), MSVC++, Digital Mars, Borland C++ 5.5 and Open Watcom compilers. Offers syntax highlighting (customizable and extensible), code folding, tabbed interface, code completion, class browser, smart indent and a To-do list management with different users and more.

- http://www.bloodshed.net/devcpp.html

Dev-C++, a free IDE including a distribution of MinGW. Delphi and C source code available.

- http://wxdsgn.sourceforge.net/

wxDev-C++, an IDE/RAD tool resulting from extending *Dev-C++*. With all the features of the previous plus others. Uses GCC for the compiler, and adds an IDE and a form designer supporting wxWidgets.

- http://quincy.codecutter.org/

Quincy 2005, a simple IDE for C and C++ under Windows. Installs the MinGW compiler and GDB debugger. Designed as a friendly learning environment. Public domain C++ source code.

- http://www.delorie.com/djgpp/

Djgpp, a free compiler for C, C++, Forth, Pascal and more including C sources. Runs under DOS.

- http://www.digitalmars.com

Digital Mars, a free C and C++ Compiler for DOS, Win & NT by the author of Zortech C++.

- http://developer.apple.com/tools/mpw-tools/

Macintosh Programmer's Workshop (MPW). Same software and documentation as the "Tool Chest:Development Kits:MPW etc." folder on the August 2001 Developer CD.

- http://www.openwatcom.org/

OpenWatcom, the Open Watcom is a joint effort between SciTech Software, Sybase®, and a select team of developers, which will bring the Sybase Watcom C, C++ and Fortran compiler products to the Open Source community.

- http://msdn.microsoft.com/mobility/prodtechinfo/devtools/eVisualc/default.aspx

Microsoft eMbedded Visual C++ allows you to develop for Windows CE. It includes an IDE, which includes an integrated debugger.

- http://www.borland.com/products/downloads/download_cbuilder.html

Borland C++Builder v5.5

- http://www.eclipse.org/

Eclipse, a multi-language IDE with support for C++ through the CDT plugin. It requires a GCC backend. There is a download specifically for C++ developers that does not include the Java libs.

8.2.2 Commercial

- http://www.intel.com/software/products/compilers/

Intel Compiler, a Intel® compilers. Compatible with the tools developers use, Intel compilers plug into popular development environments and feature source and binary compatibility with widely-used compilers. Every compiler purchase includes one year of Intel® Premier Support, providing updates, technical support and expertise for the Intel® architecture. [Intel CPUs ONLY]

- http://comeaucomputing.com/

Comeau C/C++ Compiler. It is closest to the C++ Standard. Available for purchase Comeau C/C++ supports Core C++03 language enhancements for all major and minor features of C++ and C, including export.

8.3 Misc. C++ Tools

8.3.1 *Free* or with a free version

- http://www.stack.nl/~dimitri/doxygen/

Doxygen is a documentation system for C++, C, and other programming languages.

- http://valgrind.kde.org/

Valgrind, a system for debugging and profiling applications at runtime. The system runs on nearly any x86 linux (sorry, no amd64 yet). It can detect memory

leaks, illegal memory access, double deletes, cache misses, code coverage and much, much more.

- http://msdn.microsoft.com/visualc/vctoolkit2003/

Microsoft Visual C++ Toolkit 2003, This is a free optimizing compiler provided by Microsoft that developers can use to develop and compile applications in C or C++. It is the same compiler that ships with the professional edition of Visual Studio. It ships with the standard library and sample code.

- http://ccbuild.sourceforge.net/?page=home

ccbuild, a C++ source scanning build utility for code distributed over directories. Like a dynamic Makefile, ccbuild finds all programs in the current directory (containing "int main") and builds them. For this, it reads the C++ sources and looks at all local and global includes. All C++ files surrounding local includes are considered objects for the main program. The global includes lead to extra compiler arguments using a configuration file. Next to running g++ it can create simple Makefiles, A-A-P files, and graph dependencies using DOT (Graphviz) graphs. (Linux only)

8.4 LIBRARIES[4]

8.4.1 Free or with free versions

- http://www.boost.org/

The Boost web site. Boost is a large collection of high-quality libraries for C++, some of which are likely to be included in future C++ standards.

- http://www.samblackburn.com/wfc/index.html/

The WFC (Win32 Foundation Classes) site.

- http://sourceforge.net/projects/wtl/

4 Chapter 6.3.3 on page 584

The WTL site.

- http://www.oonumerics.org/blitz/

Blitz++ is a C++ class library for scientific computing which provides performance on par with Fortran 77/90. It uses template techniques to achieve high performance. The current versions provide dense arrays and vectors, random number generators, and small vectors and matrices. Blitz++ is distributed freely under an open source LICENSE[5], and contributions to the library are welcomed.

- http://www.bdsoft.com/tools/stlfilt.html

STLFilt is a STL Error Message Decryptor for C++. It simplifies and/or reformats long-winded C++ error and warning messages, with a focus on STL-related diagnostics.

- http://www.swox.com/gmp/

GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers. There is no practical limit to the precision except the ones implied by the available memory in the machine GMP runs on.

- http://www.cryptopp.com/

Crypto++ Library is a free C++ class library of cryptographic schemes.

- http://alleg.sourceforge.net/

Allegro is a game programming library for C/C++ developers distributed FREELY[6], supporting the following platforms: DOS, Unix (Linux, FreeBSD, Irix, Solaris, Darwin), Windows, QNX, BeOS and MacOS X. It provides many functions for graphics, sounds, player input (keyboard, mouse and joystick) and

5 HTTP://WWW.OONUMERICS.ORG/BLITZ/LEGAL/
6 HTTP://ALLEG.SOURCEFORGE.NET/LICENSE.HTML

timers. It also provides fixed and floating point mathematical functions, 3d functions, file management functions, compressed datafile and a GUI.

- http://fltk.org/

FLTK (pronounced "fulltick") is a cross-platform C++ GUI toolkit for UNIX®/Linux® (X11), Microsoft® Windows®, and MacOS® X. FLTK provides modern GUI functionality without the bloat and supports 3D graphics via OpenGL® and its built-in GLUT emulation. FLTK is designed to be small and modular enough to be statically linked, but works fine as a shared library. FLTK also includes an excellent UI builder called FLUID that can be used to create applications in minutes.

- http://www.libsdl.org/

Simple DirectMedia Layer is a cross-platform multimedia library for C/C++. It provides low-level acces to 2D frame-buffer and hardware accelerated 3D graphics(using OpenGL), audio, threads, timers,user input and event handling. Other features are available through "plug-in libraries". Linux, Windows, BeOS, MacOS Classic, MacOS X, FreeBSD, OpenBSD, BSD/OS, Solaris, IRIX, and QNX are supported and there's some unofficial support for other platforms. SDL is available under the GNU LGPL[8] license.

- http://www.hpcfactor.com/developer/

SDKs for older platforms and from third parties. Includes a redistributable that contains the MFC library's for Windows CE 1, 2, HPC Pro, Palm-Size PC 1.2, HPC2000 and a limited number from Windows CE 4.0.

- http://qt.nokia.com

Qt (pronounced "cute"), a multi-platform API that contains a UI toolkit as well as a core library. It is extremely modular, but to use it effectively, you should use at least UI+Core.

- http://loki-lib.sourceforge.net/

8 HTTP://WWW.GNU.ORG/COPYLEFT/LESSER.HTML

Loki is a C++ library which demonstrates and encourages the use of generic programming and design patterns. It was written to accompany the book entitled "Modern C++ Design." The library includes a parametrized smart pointer class, generalized functors, a multi-threading abstraction, and some help for important patterns. Open source released under the MIT license.

- http://tinythread.sourceforge.net/

TinyThread++, a light weight, portable C++ thread library that implements a subset of the C++0x standard, including the thread, mutex and condition_variable classes. Open source, released under the zlib/libpng License.

8.5 C++ Coding Conventions

8.5.1 Source Code Formatting rules

- http://www.cs.usyd.edu.au/~scilect/tpop/handouts/Style.htm

Kernighan and Ritchie (or K&R) style

- http://www.nongnu.org/style-guide/

GNU Programmer's Style Guide

- http://lxr.linux.no/source/Documentation/CodingStyle

Linux kernel coding style

8.5.2 Comprehensive Source Code Convention guidelines

- http://quantlib.org/style.shtml

QuantLib Programming Style Guidelines

- http://www.research.att.com/~bs/bs_faq2.html

Bjarne Stroustrup's C++ Style and Technique FAQ

- http://www.artima.com/intv/goldilocks.html

The C++ Style Sweet Spot A Conversation with Bjarne Stroustrup, Part I by Bill Venners

- http://developer.kde.org/documentation/other/binarycompatibility.html

KDE Binary Compatibility Issues With C++

- http://www.mozilla.org/hacking/portable-cpp.html

C++ portability guide version 0.8 originally by David Williams, 27 March 1998

- http://www.chris-lott.org/resources/cstyle/Ellemtel-rules-mm.html

Programming in C++, Rules and Recommendations by FN/Mats Henricson and Erik Nyquist

- http://www.chris-lott.org/resources/cstyle/Wildfire-C++Style.html

Wildfire C++ Programming Style With Rationale by Keith Gabryelski

- http://www.kuro5hin.org/story/2002/5/9/205040/3918

Musings on Good C++ Style (Technology) by GoingWare

- http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

Google C++ Style Guide

- https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637

CERT C++ Secure Coding Standard

- http://www.research.att.com/~bs/JSF-AV-rules.pdf

Joint Strike Fighter air vehicle: C++ coding standards 2005

- http://www.chris-lott.org/resources/cstyle/

C and C++ Style Guides by Chris Lott, lists many popular C++ style guides.

- http://www.misra-cpp.org/

MISRA C++: Guidelines for the use of the C++ language in critical systems published by The Motor Industry Software Reliability Association (MISRA[9]) (based on a subset of C++).

- http://freeworld.thc.org/root/phun/unmaintain.html

A very funny satiric text that turns the tables on the issues concerning coding style by Roedy Green.

10

8.6 Online C++ books, guides and general information

- MORE C++ IDIOMS[11], covering modern C++ idioms.
- WINDOWS PROGRAMMING[12], a Wikibook on Windows API (C and VB Classic), MFC (C++), COM and creation of ActiveX modules.
- OPTIMIZING C++[13], covering how C++ programmers can improve a program's performance.

9 HTTP://EN.WIKIPEDIA.ORG/WIKI/MISRA
10 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
11 HTTP://EN.WIKIBOOKS.ORG/WIKI/MORE%20C%2B%2B%20IDIOMS
12 HTTP://EN.WIKIBOOKS.ORG/WIKI/WINDOWS%20PROGRAMMING
13 HTTP://EN.WIKIBOOKS.ORG/WIKI/OPTIMIZING%20C%2B%2B

- LINUX APPLICATIONS DEBUGGING TECHNIQUES[14], a hands-on guide to debugging applications under Linux.
- C++ IN ACTION[15], by Bartosz Milewski
- TEACH YOURSELF C++ IN 21 DAYS, SECOND EDITION[16]
- MORE C++[17], by Tim Love, July 5, 2001
- INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING USING C++[18], by Peter Müller, 1997.
- C++ PROGRAMMING FOR SCIENTISTS[19], by Roldan Pozo and Karin Remington
- C++ FOR UNIX[20], a broken link, a quick reference, with C variations
- STL QUICK REFERENCE[21], probably by Pablo Halpern,
- C++ A DIALOG[22], by Steve Heller
- LEARNING C++: AN INDEX OF ENTRY POINTS[23]
- C++ PROGRAMMING HOW-TO[24] PDF, by Al Dev (Alavoor Vasudevan), 2001
- THINK LIKE A COMPUTER SCIENTIST: C++[25]

References to other works that can be relevant to the topic:

- KOHL ET AL. 2004: C/C++ REFERENCE[26] at cppreference.com
- THE C++ ANNOTATIONS FOR C PROGRAMMERS[27], by Frank B. Brokken
- C / C++ TUTORIALS[28] at pickatutorial.com, a collection of online C / C++ tutorials

---

14 HTTP://EN.WIKIBOOKS.ORG/WIKI/LINUX%20APPLICATIONS%20DEBUGGING%20TECHNIQUES

15 HTTP://WWW.RELISOFT.COM/BOOK/INDEX.HTM

16 HTTP://WEB.ARCHIVE.ORG/WEB/20070317030353/HTTP://GUIDES.OERNII.SK/C++/INDEX.HTM

17 HTTP://WWW-H.ENG.CAM.AC.UK/HELP/TPL/LANGUAGES/C++/DOC/DOC.HTML

18 HTTP://WEB.ARCHIVE.ORG/WEB/20071030201428/HTTP://WWW.ZIB.DE/VISUAL/PEOPLE/MUELLER/COURSE/TUTORIAL/TUTORIAL.HTML

19 HTTP://MATH.NIST.GOV/~{}RPOZO/C++CLASS/

20 HTTP://WWW.CS.JCU.EDU.AU/~{}DAVID/C++SYNTAX.HTML

21 HTTP://WWW.HALPERNWIGHTSOFTWARE.COM/STDLIB-SCRATCH/QUICKREF.HTML

22 HTTP://WWW.STEVEHELLER.COM/CPPAD/OUTPUT/DIALOGTOC.HTML

23 HTTP://WEB.ARCHIVE.ORG/WEB/20080508040722/HTTP://CS.NMHU.EDU/PERSONAL/CURTIS/CS1HTMLFILES/CS1TEXT6-2001.HTM

24 HTTP://WWW.DIGILIFE.BE/QUICKREFERENCES/BOOKS/C++%20PROGRAMMING%20HOW-TO.PDF

25 HTTP://WEB.ARCHIVE.ORG/WEB/20071016111543/HTTP://WWW.IBIBLIO.ORG/OBP/THINKCS/CPP/ENGLISH/

26 HTTP://WWW.CPPREFERENCE.COM/

27 HTTP://WWW.ICCE.RUG.NL/DOCUMENTS/CPLUSPLUS/

28 HTTP://WWW.PICKATUTORIAL.COM/TUTORIALS/C_C_PLUSPLUS_1.HTM

- CPLUSPLUS.COM[29], an open resource with various web discussion groups
- A WEB SITE OF SCOTT MEYERS[30], an expert on C++ software development. He wrote the best-selling Effective C++ series (Effective C++, More Effective C++, and Effective STL), wrote and designed Effective C++ CD
- CPROGRAMMING.COM[31], a web site designed to help you learn C or C++ and provide you with C and C++ programming resources.
- MISC. BOOKS, NEWS & ARTICLES ON C++[32], by orelily.com
- FREQUENTLY ASKED QUESTIONS ABOUT WIN32 PROGRAMMING[33], from iseran.com
- CPPHERESY[34] at c2.com, guidelines about how to keep the C++ bits simple
- C++ FAQ[35] at parashift.com, sometimes also called *C++ FAQ Lite*.
- DEFECTIVE C++[36]: a 2009 summary by Yossi Kreinin of the major defects he finds in the C++ programming language. (Related: PARTICULARITIES OF THE C PROGRAMMING LANGUAGE[37]).

8.6.1 IRC

- #C at (irc://irc.tambov.ru)

C/C++ Channel (HTTP://SILVERSOFT.NET/)[38] (Russian Channel)

- #C++ at (irc://hub.ptnet.org)

C++ Channel (Portuguese Channel)

- ##C++ at (irc://irc.freenode.net)

C++ Channel

29 HTTP://WWW.CPLUSPLUS.COM/MAIN.HTML
30 HTTP://WWW.ARISTEIA.COM/
31 HTTP://WWW.CPROGRAMMING.COM/
32 HTTP://CPROG.OREILLY.COM/
33 HTTP://WWW.ISERAN.COM/WIN32/FAQ/
34 HTTP://C2.COM/CGI/WIKI?CPPHERESY
35 HTTP://WWW.PARASHIFT.COM/C++-FAQ-LITE/
36 HTTP://YOSEFK.COM/C++FQA/DEFECTIVE.HTML
37 HTTP://EN.WIKIBOOKS.ORG/WIKI/C%20PROGRAMMING%2FPARTICULARITIES%20OF%20C
38 HTTP://SILVERSOFT.NET/)

- #c++newbie at (irc://irc.freenode.net)

Channel for those new to C++

- #c++ at (irc://irc.dynastynet.net)

Channel on DynastyNet for discussing C++ topics.

8.6.2 User Groups

- http://www.accu.org/

ACCU, formerly the Association for C and C++ Users, ACCU is a non-profit organization devoted to professionalism in programming at all levels. Although primarily focused on C and C++, have now interests in Java, C# and Python also.

8.6.3 Newsgroups (NNTP)

- COMP.STD.C++[39] - FAQ[40]
- COMP.LANG.C++.LEDA[41]
- COMP.LANG.C++.MODERATED[42]
- COMP.LANG.C++[43]
- MICROSOFT.PUBLIC.VC.MFC[44]
- MICROSOFT.PUBLIC.VC.STL[45]

8.6.4 Blogs and Wikis

- http://www.codepedia.com/1/Cpp

39 HTTP://GROUPS.GOOGLE.COM/GROUP/COMP.STD.C++
40 HTTP://WWW.COMEAUCOMPUTING.COM/CSC/FAQ.HTML
41 HTTP://GROUPS.GOOGLE.COM/GROUP/COMP.LANG.C++.LEDA
42 HTTP://GROUPS.GOOGLE.COM/GROUP/COMP.LANG.C++.MODERATED
43 HTTP://GROUPS.GOOGLE.COM/GROUP/COMP.LANG.C++
44 HTTP://GROUPS.GOOGLE.COM/GROUP/MICROSOFT.PUBLIC.VC.MFC
45 HTTP://GROUPS.GOOGLE.COM/GROUP/MICROSOFT.PUBLIC.VC.STL

a Wikipedia-like page with much code examples.

- http://www.gnacademy.org/twiki/bin/view/CPP/TableOfContents%20GNAcademy.Org

TWiki C++ Web is a C++ wiki with GNU Free Documentation License

- http://cpp.wikia.com/

Wikicities C++ is a multi-language C++ wiki (currently English and Polish).

8.6.5 Mailing Lists

- http://www.oonumerics.org/mailman/listinfo.cgi/oon-list/

Object-Oriented Numerics List, forum for discussing scientific computing in object-oriented environments. An archive is AVAILABLE[46].

8.6.6 Forums

- http://stackoverflow.com/questions/tagged/c%2b%2b

Stackoverflow is a knowledge sharing community including discussing C++ related topics.

46 HTTP://WWW.OONUMERICS.ORG/MAILARCHIVES/OON-LIST/

8.7 Other (dead tree) books on C++

8.7.1 Introductory books

- Thinking in C++, 2nd ed. Volume 1: Introduction to Standard C++ by Bruce Eckel, ISBN 0139798099 | ISBN-13: 978-0139798092. Available ONLINE[47] and also as FREE DOWNLOAD[48] PDF.

8.7.2 Advanced topics

- Thinking in C++, 2nd ed. Volume 2: Practical Programming by Bruce Eckel, 0139798099 | ISBN-13: 978-0139798092. Available online as FREE DOWNLOAD[49] PDF.
- Effective C++ : 55 Specific Ways to Improve Your Programs and Designs, 3rd ed. by Scott Meyers, ISBN 0321334876

8.7.3 Reference books

- C++ in a Nutshell by Ray Lischner, ISBN 059600298X
- C++ Pocket Reference by Kyle Loudon, ISBN 0596004966
- C++ FAQs[50] by Marshall Cline, Greg Lomow, and Mike Girou, Addison-Wesley, 1999, ISBN 0-201-30983-1

[51]

.}[52]

47 HTTP://WWW.SMART2HELP.COM/E-BOOKS/TICPP-2ND-ED-VOL-ONE/FRAMES.HTML
48 HTTP://WWW.LIB.RU.AC.TH/DOWNLOAD/E-BOOKS/TIC2VONE.PDF
49 HTTP://WWW.LIB.RU.AC.TH/DOWNLOAD/E-BOOKS/TIC2VTWO.PDF
50 HTTP://PARASHIFT.COM/C++-FAQ-LITE/FAQ-BOOK.HTML
51 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING
52 HTTP://EN.WIKIBOOKS.ORG/WIKI/CATEGORY%3AC%2B%2B%20PROGRAMMING

9 Contributors

Edits	User
1	1EXEC1[1]
4	32TO28[2]
4	A.K.KARTHIKEYAN[3]
1	AK7[4]
1	ADRILEY[5]
1	ADAM MAJEWSKI[6]
2	ADIKASHI[7]
183	ADRIGNOLA[8]
6	AJM1205[9]
3	AKILAA[10]
2	ALANUS[11]
1	ALBERTCAHALAN[12]
1	ALCA ISILON[13]
1	ALEKSEV[14]
1	ALEXANDERSWANG[15]
2	ALSOCAL[16]
3	AMIN.AJANI17[17]

1 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:1EXEC1
2 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:32TO28
3 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:A.K.KARTHIKEYAN
4 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:AK7
5 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:ADRILEY
6 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:ADAM_MAJEWSKI
7 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:ADIKASHI
8 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:ADRIGNOLA
9 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:AJM1205
10 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:AKILAA
11 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:ALANUS
12 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:ALBERTCAHALAN
13 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:ALCA_ISILON
14 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:ALEKSEV
15 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:ALEXANDERSWANG
16 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:ALSOCAL
17 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:AMIN.AJANI17

1 Andre Engels[18]
2 Anunnakki[19]
3 Argento[20]
9 Arlen22[21]
3 Ashutosh.ukey[22]
4 Athgorn[23]
2 Atrium[24]
7 Autumnfields[25]
23 Avicennasis[26]
9 Az1568[27]
2 BCG999[28]
1 Bazkie botsauto[29]
2 BenFrantzDale[30]
2 Beuc[31]
1 Billymac00[32]
1 BixoDePalha[33]
1 Boggie[34]
3 Bombe[35]
1 Borb[36]
1 Bxj[37]
2 Carl Turner[38]

18 http://en.wikibooks.org/w/index.php?title=User:Andre_Engels
19 http://en.wikibooks.org/w/index.php?title=User:Anunnakki
20 http://en.wikibooks.org/w/index.php?title=User:Argento
21 http://en.wikibooks.org/w/index.php?title=User:Arlen22
22 http://en.wikibooks.org/w/index.php?title=User:Ashutosh.ukey
23 http://en.wikibooks.org/w/index.php?title=User:Athgorn
24 http://en.wikibooks.org/w/index.php?title=User:Atrium
25 http://en.wikibooks.org/w/index.php?title=User:Autumnfields
26 http://en.wikibooks.org/w/index.php?title=User:Avicennasis
27 http://en.wikibooks.org/w/index.php?title=User:Az1568
28 http://en.wikibooks.org/w/index.php?title=User:BCG999
29 http://en.wikibooks.org/w/index.php?title=User:Bazkie_botsauto
30 http://en.wikibooks.org/w/index.php?title=User:BenFrantzDale
31 http://en.wikibooks.org/w/index.php?title=User:Beuc
32 http://en.wikibooks.org/w/index.php?title=User:Billymac00
33 http://en.wikibooks.org/w/index.php?title=User:BixoDePalha
34 http://en.wikibooks.org/w/index.php?title=User:Boggie
35 http://en.wikibooks.org/w/index.php?title=User:Bombe
36 http://en.wikibooks.org/w/index.php?title=User:Borb
37 http://en.wikibooks.org/w/index.php?title=User:Bxj
38 http://en.wikibooks.org/w/index.php?title=User:Carl_Turner

1 CarsracBot[39]
1 Catsarecool[40]
1 Chesemonkyloma[41]
1 Chricho[42]
3 Chris.Seedyk[43]
13 Cleos[44]
1 Cloudguitar[45]
1 CommonsDelinker[46]
14 Coppro[47]
5 Costanzo[48]
1 Cowtung[49]
1 Crusadeonilliteracy[50]
2 Cyp[51]
4 DWarrior[52]
1 Dallas1278[53]
7 Dan Polansky[54]
2 Danilo.Piazzalunga[55]
242 Darklama[56]
49 DavidCary[57]
9 Derbeth[58]
1 Devourer09[59]

39 http://en.wikibooks.org/w/index.php?title=User:CarsracBot
40 http://en.wikibooks.org/w/index.php?title=User:Catsarecool
41 http://en.wikibooks.org/w/index.php?title=User:Chesemonkyloma
42 http://en.wikibooks.org/w/index.php?title=User:Chricho
43 http://en.wikibooks.org/w/index.php?title=User:Chris.Seedyk
44 http://en.wikibooks.org/w/index.php?title=User:Cleos
45 http://en.wikibooks.org/w/index.php?title=User:Cloudguitar
46 http://en.wikibooks.org/w/index.php?title=User:CommonsDelinker
47 http://en.wikibooks.org/w/index.php?title=User:Coppro
48 http://en.wikibooks.org/w/index.php?title=User:Costanzo
49 http://en.wikibooks.org/w/index.php?title=User:Cowtung
50 http://en.wikibooks.org/w/index.php?title=User:Crusadeonilliteracy
51 http://en.wikibooks.org/w/index.php?title=User:Cyp
52 http://en.wikibooks.org/w/index.php?title=User:DWarrior
53 http://en.wikibooks.org/w/index.php?title=User:Dallas1278
54 http://en.wikibooks.org/w/index.php?title=User:Dan_Polansky
55 http://en.wikibooks.org/w/index.php?title=User:Danilo.Piazzalunga
56 http://en.wikibooks.org/w/index.php?title=User:Darklama
57 http://en.wikibooks.org/w/index.php?title=User:DavidCary
58 http://en.wikibooks.org/w/index.php?title=User:Derbeth
59 http://en.wikibooks.org/w/index.php?title=User:Devourer09

26 Dirk Hünniger[60]
9 Dnas[61]
1 Dopple[62]
2 Dresdnhope[63]
1 DuckMan21[64]
1 Duncanphilipnorman[65]
1 Dvir.Kafri[66]
6 Dysprosia[67]
2 Dzajtai[68]
1 E James[69]
1 Edudobay[70]
2 ElieDeBrauwer[71]
1 Emanuele[72]
7 Emperorbma[73]
2 Emry[74]
1 EncMstr[75]
1 EphemeralJun[76]
1 Esben[77]
1 Fasten[78]
5 Faulknerck2[79]
4 Fd0man[80]

60 http://en.wikibooks.org/w/index.php?title=User:Dirk_H%C3%BCnniger
61 http://en.wikibooks.org/w/index.php?title=User:Dnas
62 http://en.wikibooks.org/w/index.php?title=User:Dopple
63 http://en.wikibooks.org/w/index.php?title=User:Dresdnhope
64 http://en.wikibooks.org/w/index.php?title=User:DuckMan21
65 http://en.wikibooks.org/w/index.php?title=User:Duncanphilipnorman
66 http://en.wikibooks.org/w/index.php?title=User:Dvir.Kafri
67 http://en.wikibooks.org/w/index.php?title=User:Dysprosia
68 http://en.wikibooks.org/w/index.php?title=User:Dzajtai
69 http://en.wikibooks.org/w/index.php?title=User:E_James
70 http://en.wikibooks.org/w/index.php?title=User:Edudobay
71 http://en.wikibooks.org/w/index.php?title=User:ElieDeBrauwer
72 http://en.wikibooks.org/w/index.php?title=User:Emanuele
73 http://en.wikibooks.org/w/index.php?title=User:Emperorbma
74 http://en.wikibooks.org/w/index.php?title=User:Emry
75 http://en.wikibooks.org/w/index.php?title=User:EncMstr
76 http://en.wikibooks.org/w/index.php?title=User:EphemeralJun
77 http://en.wikibooks.org/w/index.php?title=User:Esben
78 http://en.wikibooks.org/w/index.php?title=User:Fasten
79 http://en.wikibooks.org/w/index.php?title=User:Faulknerck2
80 http://en.wikibooks.org/w/index.php?title=User:Fd0man

1 Ferox[81]
55 Fishpi[82]
2 Francis Ocoma[83]
3 Fredo[84]
1 FunPika[85]
1 Galoubet[86]
10 Garrett[87]
2 Gentgeen[88]
1 Geocachernemesis[89]
1 GeordieMcBain[90]
1 George Hernandez[91]
90 Ghfjdk[92]
4 Ghostzart[93]
8 Gmcfoley[94]
2 GoToMan[95]
1 Gopalakrishnans[96]
2 Graeme[97]
1 Grawp[98]
1 Grawp the Giant[99]
1 Green caterpillar[100]

81 http://en.wikibooks.org/w/index.php?title=User:Ferox
82 http://en.wikibooks.org/w/index.php?title=User:Fishpi
83 http://en.wikibooks.org/w/index.php?title=User:Francis_Ocoma
84 http://en.wikibooks.org/w/index.php?title=User:Fredo
85 http://en.wikibooks.org/w/index.php?title=User:FunPika
86 http://en.wikibooks.org/w/index.php?title=User:Galoubet
87 http://en.wikibooks.org/w/index.php?title=User:Garrett
88 http://en.wikibooks.org/w/index.php?title=User:Gentgeen
89 http://en.wikibooks.org/w/index.php?title=User:Geocachernemesis
90 http://en.wikibooks.org/w/index.php?title=User:GeordieMcBain
91 http://en.wikibooks.org/w/index.php?title=User:George_Hernandez
92 http://en.wikibooks.org/w/index.php?title=User:Ghfjdk
93 http://en.wikibooks.org/w/index.php?title=User:Ghostzart
94 http://en.wikibooks.org/w/index.php?title=User:Gmcfoley
95 http://en.wikibooks.org/w/index.php?title=User:GoToMan
96 http://en.wikibooks.org/w/index.php?title=User:Gopalakrishnans
97 http://en.wikibooks.org/w/index.php?title=User:Graeme
98 http://en.wikibooks.org/w/index.php?title=User:Grawp
99 http://en.wikibooks.org/w/index.php?title=User:Grawp_the_Giant
100 http://en.wikibooks.org/w/index.php?title=User:Green_caterpillar

2 GreenVoid[101]
11 Gronau[102]
1 Grumbel[103]
1 Guanaco[104]
1 Gurupathi[105]
1 Gwylim a[106]
5 Hagindaz[107]
1 Hammerjw[108]
1 Hao2lian[109]
55 Herbythyme[110]
10 HereToHelp[111]
2 HethrirBot[112]
1 Hybridpro[113]
1 IBB[114]
1 IO[115]
2 Iamunknown[116]
38 Ikarsik[117]
2 IlyaHaykinson[118]
7 Invader02[119]
1 Ixtli[120]
3 James Brown[121]

101 http://en.wikibooks.org/w/index.php?title=User:GreenVoid
102 http://en.wikibooks.org/w/index.php?title=User:Gronau
103 http://en.wikibooks.org/w/index.php?title=User:Grumbel
104 http://en.wikibooks.org/w/index.php?title=User:Guanaco
105 http://en.wikibooks.org/w/index.php?title=User:Gurupathi
106 http://en.wikibooks.org/w/index.php?title=User:Gwylim_a
107 http://en.wikibooks.org/w/index.php?title=User:Hagindaz
108 http://en.wikibooks.org/w/index.php?title=User:Hammerjw
109 http://en.wikibooks.org/w/index.php?title=User:Hao2lian
110 http://en.wikibooks.org/w/index.php?title=User:Herbythyme
111 http://en.wikibooks.org/w/index.php?title=User:HereToHelp
112 http://en.wikibooks.org/w/index.php?title=User:HethrirBot
113 http://en.wikibooks.org/w/index.php?title=User:Hybridpro
114 http://en.wikibooks.org/w/index.php?title=User:IBB
115 http://en.wikibooks.org/w/index.php?title=User:IO
116 http://en.wikibooks.org/w/index.php?title=User:Iamunknown
117 http://en.wikibooks.org/w/index.php?title=User:Ikarsik
118 http://en.wikibooks.org/w/index.php?title=User:IlyaHaykinson
119 http://en.wikibooks.org/w/index.php?title=User:Invader02
120 http://en.wikibooks.org/w/index.php?title=User:Ixtli
121 http://en.wikibooks.org/w/index.php?title=User:James_Brown

154	James Dennett[122]
1	JamesCrook[123]
1	Jamesofur[124]
2	Jayaram Ganapathy[125]
1	Jaydeepmehtawiki[126]
3	JeffSchwab1[127]
1	Jeremyroman[128]
13	Jfmantis[129]
57	Jguk[130]
1	Jleedev[131]
2	Jlenthe[132]
1	Joecool94[133]
1	JohnOwens[134]
1	Johnruble[135]
1	Jokes Free4Me[136]
5	Jomegat[137]
1	Jonas Nordlund[138]
1	Jorgenev[139]
1	João Jerónimo[140]
17	Juxe[141]

122 http://en.wikibooks.org/w/index.php?title=User:James_Dennett
123 http://en.wikibooks.org/w/index.php?title=User:JamesCrook
124 http://en.wikibooks.org/w/index.php?title=User:Jamesofur
125 http://en.wikibooks.org/w/index.php?title=User:Jayaram_Ganapathy
126 http://en.wikibooks.org/w/index.php?title=User:Jaydeepmehtawiki
127 http://en.wikibooks.org/w/index.php?title=User:JeffSchwab1
128 http://en.wikibooks.org/w/index.php?title=User:Jeremyroman
129 http://en.wikibooks.org/w/index.php?title=User:Jfmantis
130 http://en.wikibooks.org/w/index.php?title=User:Jguk
131 http://en.wikibooks.org/w/index.php?title=User:Jleedev
132 http://en.wikibooks.org/w/index.php?title=User:Jlenthe
133 http://en.wikibooks.org/w/index.php?title=User:Joecool94
134 http://en.wikibooks.org/w/index.php?title=User:JohnOwens
135 http://en.wikibooks.org/w/index.php?title=User:Johnruble
136 http://en.wikibooks.org/w/index.php?title=User:Jokes_Free4Me
137 http://en.wikibooks.org/w/index.php?title=User:Jomegat
138 http://en.wikibooks.org/w/index.php?title=User:Jonas_Nordlund
139 http://en.wikibooks.org/w/index.php?title=User:Jorgenev
140 http://en.wikibooks.org/w/index.php?title=User:Jo%C3%A3o_Jer%C3%B3nimo
141 http://en.wikibooks.org/w/index.php?title=User:Juxe

5 KTC[142]
1 KAKURADY[143]
6 KATE[144]
7 KAYAU[145]
1 KJETIL R[146]
16 KRISCHIK[147]
20 LEANDROGOE[148]
3 LEMMIO[149]
1 LINUXFREAK[150]
1 MRPROGRAMMER[151]
5 MVHOKIES[152]
5 MAHANGA[153]
2 MARCELO PINTO[154]
1 MARCUS256[155]
1 MARKHUDSON[156]
1 MARTNYM[157]
2 MASLEN[158]
21 MATHWIZARD1232[159]
2 MATTB112885[160]
1 MATTIEUGA[161]
47 MAXBERGER[162]

142 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:KTC
143 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:KAKURADY
144 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:KATE
145 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:KAYAU
146 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:KJETIL_R
147 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:KRISCHIK
148 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:LEANDROGOE
149 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:LEMMIO
150 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:LINUXFREAK
151 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:MRPROGRAMMER
152 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:MVHOKIES
153 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:MAHANGA
154 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:MARCELO_PINTO
155 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:MARCUS256
156 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:MARKHUDSON
157 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:MARTNYM
158 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:MASLEN
159 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:MATHWIZARD1232
160 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:MATTB112885
161 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:MATTIEUGA
162 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:MAXBERGER

1 McIntosh Natura[163]
32 Merrheim[164]
1 Michaeldadmum[165]
2 Mike.lifeguard[166]
1 Mikel[167]
2 Miklcct[168]
1 Mital d vora[169]
32 Mjchael[170]
3 Mkn[171]
1 Morningstar2651[172]
1 Mrajcok[173]
30 Mshonle[174]
1 Msiladin[175]
1 Mskonline[176]
5 N.hariharan 1988[177]
2 N313t3[178]
8 Neuviemep[179]
1 Nicholasbishop[180]
2 NickWhaleyIsSexy[181]
2 Nikiriy[182]
2 NipplesMeCool[183]

163 http://en.wikibooks.org/w/index.php?title=User:McIntosh_Natura
164 http://en.wikibooks.org/w/index.php?title=User:Merrheim
165 http://en.wikibooks.org/w/index.php?title=User:Michaeldadmum
166 http://en.wikibooks.org/w/index.php?title=User:Mike.lifeguard
167 http://en.wikibooks.org/w/index.php?title=User:Mikel
168 http://en.wikibooks.org/w/index.php?title=User:Miklcct
169 http://en.wikibooks.org/w/index.php?title=User:Mital_d_vora
170 http://en.wikibooks.org/w/index.php?title=User:Mjchael
171 http://en.wikibooks.org/w/index.php?title=User:Mkn
172 http://en.wikibooks.org/w/index.php?title=User:Morningstar2651
173 http://en.wikibooks.org/w/index.php?title=User:Mrajcok
174 http://en.wikibooks.org/w/index.php?title=User:Mshonle
175 http://en.wikibooks.org/w/index.php?title=User:Msiladin
176 http://en.wikibooks.org/w/index.php?title=User:Mskonline
177 http://en.wikibooks.org/w/index.php?title=User:N.hariharan_1988
178 http://en.wikibooks.org/w/index.php?title=User:N313t3
179 http://en.wikibooks.org/w/index.php?title=User:Neuviemep
180 http://en.wikibooks.org/w/index.php?title=User:Nicholasbishop
181 http://en.wikibooks.org/w/index.php?title=User:NickWhaleyIsSexy
182 http://en.wikibooks.org/w/index.php?title=User:Nikiriy
183 http://en.wikibooks.org/w/index.php?title=User:NipplesMeCool

3 NithinBekal[184]
1 Nmrtian[185]
13 Noldoaran[186]
1 Nrcarballo[187]
5 Ojan[188]
7 Omair.majid[189]
52 Orderud[190]
45 Paddu[191]
5976 Panic2k4[192]
3 Phatency[193]
1 Phil.a[194]
6 Phosgram[195]
3 Pie21[196]
14 Poeticjustice712182[197]
1 Pradeep reddy[198]
1 Prasannjit.gondchawar[199]
1 Prime[200]
27 Pumbaa80[201]
2 PurplePieman[202]
2 QUBot[203]
4 Qazsedcft[204]

184 http://en.wikibooks.org/w/index.php?title=User:NithinBekal
185 http://en.wikibooks.org/w/index.php?title=User:Nmrtian
186 http://en.wikibooks.org/w/index.php?title=User:Noldoaran
187 http://en.wikibooks.org/w/index.php?title=User:Nrcarballo
188 http://en.wikibooks.org/w/index.php?title=User:Ojan
189 http://en.wikibooks.org/w/index.php?title=User:Omair.majid
190 http://en.wikibooks.org/w/index.php?title=User:Orderud
191 http://en.wikibooks.org/w/index.php?title=User:Paddu
192 http://en.wikibooks.org/w/index.php?title=User:Panic2k4
193 http://en.wikibooks.org/w/index.php?title=User:Phatency
194 http://en.wikibooks.org/w/index.php?title=User:Phil.a
195 http://en.wikibooks.org/w/index.php?title=User:Phosgram
196 http://en.wikibooks.org/w/index.php?title=User:Pie21
197 http://en.wikibooks.org/w/index.php?title=User:Poeticjustice712182
198 http://en.wikibooks.org/w/index.php?title=User:Pradeep_reddy
199 http://en.wikibooks.org/w/index.php?title=User:Prasannjit.gondchawar
200 http://en.wikibooks.org/w/index.php?title=User:Prime
201 http://en.wikibooks.org/w/index.php?title=User:Pumbaa80
202 http://en.wikibooks.org/w/index.php?title=User:PurplePieman
203 http://en.wikibooks.org/w/index.php?title=User:QUBot
204 http://en.wikibooks.org/w/index.php?title=User:Qazsedcft

7 QuiteUnusual[205]
1 Ramac[206]
1 Ramir[207]
2 Ravensky[208]
4 Recent Runes[209]
22 Remi0o[210]
3 Renich[211]
1 Res1233[212]
2 Revolus[213]
1 Rfrohardt[214]
6 Rizvn[215]
5 Rmccue[216]
1 Robchurch[217]
1 Rodasmith[218]
2 Rohitvipin[219]
20 Ronyclau[220]
2 Ruakh[221]
7 Rwdougla[222]
11 SBJohnny[223]
5 Sae1962[224]
4 Safedoctor[225]

205 http://en.wikibooks.org/w/index.php?title=User:QuiteUnusual
206 http://en.wikibooks.org/w/index.php?title=User:Ramac
207 http://en.wikibooks.org/w/index.php?title=User:Ramir
208 http://en.wikibooks.org/w/index.php?title=User:Ravensky
209 http://en.wikibooks.org/w/index.php?title=User:Recent_Runes
210 http://en.wikibooks.org/w/index.php?title=User:Remi0o
211 http://en.wikibooks.org/w/index.php?title=User:Renich
212 http://en.wikibooks.org/w/index.php?title=User:Res1233
213 http://en.wikibooks.org/w/index.php?title=User:Revolus
214 http://en.wikibooks.org/w/index.php?title=User:Rfrohardt
215 http://en.wikibooks.org/w/index.php?title=User:Rizvn
216 http://en.wikibooks.org/w/index.php?title=User:Rmccue
217 http://en.wikibooks.org/w/index.php?title=User:Robchurch
218 http://en.wikibooks.org/w/index.php?title=User:Rodasmith
219 http://en.wikibooks.org/w/index.php?title=User:Rohitvipin
220 http://en.wikibooks.org/w/index.php?title=User:Ronyclau
221 http://en.wikibooks.org/w/index.php?title=User:Ruakh
222 http://en.wikibooks.org/w/index.php?title=User:Rwdougla
223 http://en.wikibooks.org/w/index.php?title=User:SBJohnny
224 http://en.wikibooks.org/w/index.php?title=User:Sae1962
225 http://en.wikibooks.org/w/index.php?title=User:Safedoctor

1 Sameen[226]
1 Sarang[227]
6 Schwarzbichler[228]
5 Scr[229]
3 Sddoc[230]
1 Seraphimblade[231]
1 ShakespeareFan00[232]
3 Shokuku[233]
97 Sigma 7[234]
5 Sikandaramin[235]
1 Sisingh[236]
1 Sj[237]
1 Slposey[238]
3 Spartacus3d[239]
1 Spaz man[240]
11 Spoon![241]
1 Sridarshan23[242]
2 StephenMorrisson[243]
1 Sutambe[244]
2 Swift[245]
1 Sygmn[246]

226 http://en.wikibooks.org/w/index.php?title=User:Sameen
227 http://en.wikibooks.org/w/index.php?title=User:Sarang
228 http://en.wikibooks.org/w/index.php?title=User:Schwarzbichler
229 http://en.wikibooks.org/w/index.php?title=User:Scr
230 http://en.wikibooks.org/w/index.php?title=User:Sddoc
231 http://en.wikibooks.org/w/index.php?title=User:Seraphimblade
232 http://en.wikibooks.org/w/index.php?title=User:ShakespeareFan00
233 http://en.wikibooks.org/w/index.php?title=User:Shokuku
234 http://en.wikibooks.org/w/index.php?title=User:Sigma_7
235 http://en.wikibooks.org/w/index.php?title=User:Sikandaramin
236 http://en.wikibooks.org/w/index.php?title=User:Sisingh
237 http://en.wikibooks.org/w/index.php?title=User:Sj
238 http://en.wikibooks.org/w/index.php?title=User:Slposey
239 http://en.wikibooks.org/w/index.php?title=User:Spartacus3d
240 http://en.wikibooks.org/w/index.php?title=User:Spaz_man
241 http://en.wikibooks.org/w/index.php?title=User:Spoon%21
242 http://en.wikibooks.org/w/index.php?title=User:Sridarshan23
243 http://en.wikibooks.org/w/index.php?title=User:StephenMorrisson
244 http://en.wikibooks.org/w/index.php?title=User:Sutambe
245 http://en.wikibooks.org/w/index.php?title=User:Swift
246 http://en.wikibooks.org/w/index.php?title=User:Sygmn

1 Sykil[247]
1 Taiko[248]
1 Tajendra[249]
1 Tally Solleni[250]
1 Tannersf[251]
1 Tardis[252]
4 Tektonik[253]
32 Tero[254]
1 Tewy[255]
1 Tharenthel[256]
6 The Master[257]
1 The bellman[258]
72 Thenub314[259]
1 ThreeE[260]
1 Topsfield99[261]
23 Trevor Andersen[262]
1 Unforgettableid[263]
1 Uvgroovy[264]
3 Van der Hoorn[265]
2 Vixfear[266]
2 Voed[267]

247 http://en.wikibooks.org/w/index.php?title=User:Sykil
248 http://en.wikibooks.org/w/index.php?title=User:Taiko
249 http://en.wikibooks.org/w/index.php?title=User:Tajendra
250 http://en.wikibooks.org/w/index.php?title=User:Tally_Solleni
251 http://en.wikibooks.org/w/index.php?title=User:Tannersf
252 http://en.wikibooks.org/w/index.php?title=User:Tardis
253 http://en.wikibooks.org/w/index.php?title=User:Tektonik
254 http://en.wikibooks.org/w/index.php?title=User:Tero
255 http://en.wikibooks.org/w/index.php?title=User:Tewy
256 http://en.wikibooks.org/w/index.php?title=User:Tharenthel
257 http://en.wikibooks.org/w/index.php?title=User:The_Master
258 http://en.wikibooks.org/w/index.php?title=User:The_bellman
259 http://en.wikibooks.org/w/index.php?title=User:Thenub314
260 http://en.wikibooks.org/w/index.php?title=User:ThreeE
261 http://en.wikibooks.org/w/index.php?title=User:Topsfield99
262 http://en.wikibooks.org/w/index.php?title=User:Trevor_Andersen
263 http://en.wikibooks.org/w/index.php?title=User:Unforgettableid
264 http://en.wikibooks.org/w/index.php?title=User:Uvgroovy
265 http://en.wikibooks.org/w/index.php?title=User:Van_der_Hoorn
266 http://en.wikibooks.org/w/index.php?title=User:Vixfear
267 http://en.wikibooks.org/w/index.php?title=User:Voed

1 VOXHUMANA[268]
2 WEBAWARE[269]
4 WEEVIL[270]
8 WHITEKNIGHT[271]
1 WIKIWIZARD[272]
1 WIKIMI-DHIANN[273]
8 WILLOWTT[274]
1 WITHINFOCUS[275]
1 WORMSTONE[276]
8 XIONG CHIAMIOV[277]
10 XIXTAS[278]
5 XRABOHROK[279]
4 XRCHZ[280]
1 YAFINE[281]
1 YVH11A[282]
1 ZARTOM[283]
1 ZORBATHUT[284]
1 **Пика Пика**[285]

268 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:VOXHUMANA
269 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:WEBAWARE
270 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:WEEVIL
271 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:WHITEKNIGHT
272 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:WIKIWIZARD
273 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:WIKIMI-DHIANN
274 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:WILLOWTT
275 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:WITHINFOCUS
276 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:WORMSTONE
277 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:XIONG_CHIAMIOV
278 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:XIXTAS
279 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:XRABOHROK
280 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:XRCHZ
281 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:YAFINE
282 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:YVH11A
283 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:ZARTOM
284 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:ZORBATHUT
285 HTTP://EN.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=USER:%D0%9F%D0%B8%D0%BA%D0%B0_%D0%9F%D0%B8%D0%BA%D0%B0

List of Figures

- GFDL: Gnu Free Documentation License. HTTP://WWW.GNU.ORG/LICENSES/FDL.HTML
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. HTTP://CREATIVECOMMONS.ORG/LICENSES/BY-SA/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. HTTP://CREATIVECOMMONS.ORG/LICENSES/BY-SA/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. HTTP://CREATIVECOMMONS.ORG/LICENSES/BY-SA/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. HTTP://CREATIVECOMMONS.ORG/LICENSES/BY-SA/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. HTTP://CREATIVECOMMONS.ORG/LICENSES/BY/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. HTTP://CREATIVECOMMONS.ORG/LICENSES/BY/2.0/DEED.EN
- cc-by-2.5: Creative Commons Attribution 2.5 License. HTTP://CREATIVECOMMONS.ORG/LICENSES/BY/2.5/DEED.EN
- cc-by-3.0: Creative Commons Attribution 3.0 License. HTTP://CREATIVECOMMONS.ORG/LICENSES/BY/3.0/DEED.EN
- GPL: GNU General Public License. HTTP://WWW.GNU.ORG/LICENSES/GPL-2.0.TXT
- LGPL: GNU Lesser General Public License. HTTP://WWW.GNU.ORG/LICENSES/LGPL.HTML
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.

- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. HTTP://ARTLIBRE.ORG/LICENCE/LAL/DE
- CFR: Copyright free use.
- EPL: Eclipse Public License. HTTP://WWW.ECLIPSE.ORG/ORG/DOCUMENTS/EPL-V10.PHP

Copies of the GPL, the LGPL as well as a GFDL are included in chapter LICENSES[286].

286 Chapter 10 on page 679

1	JTojnar	PD
2	-	GFDL
3	JohnManuel[287]	GFDL
4	David Vignoni	LGPL
5	David Vignoni	LGPL
6	David Vignoni	LGPL
7	Original uploader was Ehamberg[288] at en.wikipedia[289] Later version(s) were uploaded by Wapcaplet[290], AzaToth[291], Heptite[292], MineralÃ¨[293], Procus the Mad[294], Byondlimits[295], Jvihavainen[296] at en.wikipedia[297].	GPL
8	User Cleos on English Wikibooks[298]	PD
9	User Cleos on English Wikibooks[299]	PD
10	User Cleos on English Wikibooks[300]	PD
11	Panic2k7	PD
12	User Cleos on English Wikibooks[301]	PD
13	User Cleos on English Wikibooks[302]	PD
14	User Cleos on English Wikibooks[303]	PD
15	User Cleos on English Wikibooks[304]	PD
16	User Cleos on English Wikibooks[305]	PD
17	User Cleos on English Wikibooks[306]	PD
18	User Cleos on English Wikibooks[307]	PD

287 http://en.wikibooks.org/wiki/User%3AJohnManuel
288 http://en.wikibooks.org/wiki/%3Aen%3AUser%3AEhamberg
289 http://en.wikipedia.org
290 http://en.wikibooks.org/wiki/%3Aen%3AUser%3AWapcaplet
291 http://en.wikibooks.org/wiki/%3Aen%3AUser%3AAzaToth
292 http://en.wikibooks.org/wiki/%3Aen%3AUser%3AHeptite
293 http://en.wikibooks.org/wiki/%3Aen%3AUser%3AMineral%C3%A8
294 http://en.wikibooks.org/wiki/%3Aen%3AUser%3AProcus%20the%20Mad
295 http://en.wikibooks.org/wiki/%3Aen%3AUser%3AByondlimits
296 http://en.wikibooks.org/wiki/%3Aen%3AUser%3AJvihavainen
297 http://en.wikipedia.org
298 http://en.wikibooks.org/wiki/User%3ACleos
299 http://en.wikibooks.org/wiki/User%3ACleos
300 http://en.wikibooks.org/wiki/User%3ACleos
301 http://en.wikibooks.org/wiki/User%3ACleos
302 http://en.wikibooks.org/wiki/User%3ACleos
303 http://en.wikibooks.org/wiki/User%3ACleos
304 http://en.wikibooks.org/wiki/User%3ACleos
305 http://en.wikibooks.org/wiki/User%3ACleos
306 http://en.wikibooks.org/wiki/User%3ACleos
307 http://en.wikibooks.org/wiki/User%3ACleos

19	Daniel B[308]	GFDL
20	Panic2k7[309]	GFDL
21	Panic2k7[310]	GFDL
22	User:TuukkaH[311]	PD
23	User:TuukkaH[312]	PD
24	Ilya Voyager[313]	GFDL
25	Paul R. McJones	GFDL
26	:en:User:Cburnett[314]	GFDL
27	:en:User:Khazadum[315], User:Stannered[316]	PD
28	AdamD[317]	PD

308 http://de.wikibooks.org/wiki/Benutzer%3ADaniel%20B
309 http://en.wikibooks.org/wiki/User%3APanic2k7
310 http://en.wikibooks.org/wiki/User%3APanic2k7
311 http://en.wikibooks.org/wiki/User%3ATuukkaH
312 http://en.wikibooks.org/wiki/User%3ATuukkaH
313 http://en.wikibooks.org/wiki/User%3AIlya%20Voyager
314 http://en.wikibooks.org/wiki/%3Aen%3AUser%3ACburnett
315 http://en.wikibooks.org/wiki/%3Aen%3AUser%3AKhazadum
316 http://en.wikibooks.org/wiki/User%3AStannered
317 http://en.wikibooks.org/wiki/User%3AAdamD

10 Licenses

10.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices". * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate

your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others’ Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

<one line to give the program’s name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’. This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, your program’s commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

10.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. * C. State on the Title page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. * K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. * N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements". 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

10.3 GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below. 0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

* a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or * b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: o 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. o 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. * e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

* a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. * b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.