
C++
Programming
Wikibooks

Table of Contents

About the book

Foreword

Guide to Readers

This is a wikibook (en.wikibooks.org), as such you should learn a bit about what it is and how it does its magic.

The book 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...

If you are already familiar with programming in other languages you can skip most of the **Getting Started Chapter** (it deals with basic and general programming concepts). You should not skip the *Programming Paradigms* introduction, since C++ does have some particulars on that topic that should be useful even if you already know an OOP language. The **Language Comparisons Section**, providing comparisons for the language(s) you already know, is important for veterans. However if this is your first contact with programming then continue on reading, and take in consideration that the *Programming Paradigms* section can be hard to digest if you lack some bases, don't despair, the relevant points will be extended when other concepts are introduced, that section is provided to give you a mental framework to help you not only to understand C++, but to let you easily adapt to (and from) other languages that share those concepts.

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 or the part 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...

Guide to Writers

Authors/Contributors (at Wikibooks) should register if intending to make non-anonymous contributions to the book (this will give more value and relevance to your opinions and views on the evolution of the work and enable others to talk to you) and try to follow the structure. If you have major ideas or big changes use the discussion area; as a rule just go with the flow...

Conventions

> A set of conventions have been adopted on the creation of this book, please read about them before you contribute any content on the book's talk page.

Authors

The following people are authors to this book

Panic

There are many other contributors/editors to the book; a verifiable list of all contributions exist as History Logs at Wikibooks (http://en.wikibooks.org/).

Acknowledgment is given for using some contents from other works like Programming C-/- -/-, Wikipedia, the Wikibooks Java Programming and C Programming, C++ Exercises for beginners, C/C++ Reference Web Site, and from Wikisource as from authors such as Scott Wheeler, Stephen Ferg and Ivor Horton.

Getting Started

Introducing C++

C++ (pronounced "see plus plus") is a general-purpose, object-oriented, statically typed, free-form, multi-paradigm programming language supporting procedural programming, data abstraction, and generic programming. During the 1990s, C++ became one of the most popular computer programming languages.

History

Bjarne Stroustrup from Bell Labs was the designer and original implementer of C++ (originally named "C with Classes") during the 1980s as an enhancement to the C programming language. Enhancements started with the addition of classes, followed by, among many features, virtual functions, operator overloading, multiple inheritance, templates, and exception handling, these and other features are covered in detail along this book.

The C++ programming language is a standard recognized by the the ANSI (The American National Standards Institute), BSI (The British Standards Institute), DIN (The German national standards organization), several other national standards bodies, and was ratified in 1998 by the ISO (The International Standards Organization) as *ISO/IEC 14882:1998*, the current version of which is the 2003 version, *ISO/IEC 14882:2003*.

The 1998 C++ Standard consists of two parts: the Core Language and the Standard Library; the latter includes the Standard Template Library and the Standard C Library (ANSI C 89). Many C++ libraries exist which are not part of the Standard, such as Boost. Also, non-Standard libraries written in C can generally be used by C++ programs.

Features introduced in C++ include declarations as statements, function-like casts, `new`/`delete`, `bool`, reference types, `const`, `inline` functions, default arguments, function overloading, namespaces, 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, and was 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).

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 offers a detailed explanation of this code; the included source code is to give you an idea of a simple C++ program.

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, while not unique to C++, help in understanding programming in general. Readers who have experience in another programming language may wish to skim through or skip 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 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.

Why learn C++?

Why not? This is the most clarifying approach to the decission to learn anything, learning is always good, selecting what to learn is more important as is how to prioritize tasks. Another side of this problem is that you will be investing some time in getting a new skill set, you have to consider in what ways will this benefit you. Check your objectives and compare similar projects or see what the programming market is in need.

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 should be best served in learning first two other languages, this will clarify what makes C++ special in its approach to programming problems. You should select one imperative language and in this C will probably have a better market value and will have a direct relation to C++ (a good substitute would be ASM) and the second language should be an Object Oriented language like Java for the same reasons, as there is a close relation between the tree languages.

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

Learning C is not a requirement for understanding C++, but knowing how to use an imperative language is, C++ will not make it easy for you to understand and distinguish some of this deeper concepts, since in C++ you are free to implement solutions with a great range of freedom. Understanding what options to make will become the cornerstone of mastering the language.

You should not learn C++ if you are only interested in applying or learning about Object Oriented programing since the nomenclature used and some of the approaches C++ takes to the problem will probably increase the difficulty level in learning and mastering those concepts, if you are truly interested in Object Oriented programming, the best language for that is Smalltalk.

As with all languages C++ has a specific scope of application, were it can truly shine, and if we take a quick comparison with the previous mentioned languages, 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 but will grant you a bigger control and responsibility than Java and will also not provide the default features you can obtain in similar higher level languages. You will have to search and examine several external implementations of these features and freely select those that best serve your purposes or even have to implement your own solution.

What is a Programming Language?

In the most basic terms, a "programming language" 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 and High-level, next we will introduce you to these concepts and their relevance to C++.

Low-level Languages

There are two general types of low level "languages".

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.

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.

High-level Languages

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

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.

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** 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.

A **compiler** is a program that translates the instruction of a program 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 is given on the C++ Programming Compiler Section 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 C++ Programming Statements Section 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 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, 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 of the book and for some of you that already know some other languages should look into Programming Languages Comparisons Section.

Programming Paradigms

A **programming paradigm** is a style or model of programming that affects the way programmers can design, organize and write programs. A multiparadigm programming language allows programmers to choose from a number of different programming paradigms. C++ is a multiparadigm programming language.

Procedural Programming

Procedural programming is a programming paradigm based upon the idea of a procedure call. Procedure calls are modular and are bound by scope. A procedural program is composed of one or more modules. Each module is composed of one or more subprograms. 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.

Object-Oriented Programming

Object-oriented programming 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. For example, a car is like an object, 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 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 don't 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 doesn't 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 (or hex color code), respectively. The class also lays out the methods for us, telling the car how to park and drive, but these methods can't take any action with just the blueprint - they need an object to have an effect.

Inheritance

This concept 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), though non subtyping inheritance is also possible.

Inheritance is typically expressed by describing *classes* of objects arranged in an *inheritance hierarchy* reflecting common behavior.

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 which 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 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 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.

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.

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*.

Dynamic polymorphism will be covered deeply in the Classes Section where we will address its use on redefining the method in the derived class.

Generic Programming

Generic programming or **polymorphism** is a programming style that emphasizes techniques that allow one value to take on different types as long as certain contracts such as subtypes and signature are kept. Templates popularized the notion of generics. Templates allow code to be written without consideration of the type with which it will eventually be used. Some Templates are defined in the Standard Template Library (STL).

Statically Typed

Typing refers to how a computer language handles its variables. As we will see in depth later, 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 optimizations. Since the compiler knows the exact types that are in use, it can produce machine code that is optimized for a limited sets of types. In C++, variables need to be defined before they are used so that compilers know what type they are.

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 an error when a type leads to ambiguity or incompatible usage. However, programmers disagree over how common type 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 distributed code that has proved reliable.

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 on the properties that can be expected from powerful language constructs. For example, indiscriminate use of recursion or global variables 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.

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 code, which is code that is purposefully written to be difficult to understand. However, complicated programming is also a security device, ensuring that the source code is harder to duplicate, short of using the whole program exactly how it was originally written.

Language Comparisons

There isn't a perfect language. It all depends on the tools and the objective. The optimal language (in terms of run-time performance) is machine code but machine code (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 section C++ will be compared with other closely related languages like C, Java, C# and C++/CLI.

«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) tools that will address part of this problem but its use does require planing and some degree of complexity.

The intention of these sections is not to promote a 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

C++ allows for optional garbage collection, rather than requiring that it be provided on all systems supporting C++. In the Garbage Collection Section of this book we will cover this issue deeply.

Why doesn't C++ include a finally keyword?

As we will see in the Resource Acquisition Is Initialization (RAII) Section 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 `File` class have to do I/O in a `try/catch/finally` block so that they can guarantee that the `File` is closed). With RAII, the destructor of the `File` class can make that guarantee. Now the cleanup code has to be coded only once — in the destructor of `File`; the users of the class don't need to do anything.

TODO
Split this explanation to RAII and only provide the reference

Mixing Languages

TODO
Add relevant information

C 89/99

C 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 and so we can even state that C was a subset of C++, most recent C++ compilers will also compile C code taking into consideration the small incompatibilities, since C99 and C++ 2003 are not compatible any more. You can also check more information about the C language on the C Programming Wikibook (http://en.wikibooks.org/wiki/C_Programming).

C++ as defined by the ANSI standard in 98 (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 there in C++. Some (including Stroustrup himself) have argued that the changes brought about in C99 have a philosophy 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 results didn't happen and it can be said that the languages started even to diverge.

Some of the differences are:

- C++ supports function overloading (absent in C89, allowed only for some standard library code in C99).
- C++ supports inheritance and polymorphism.
- 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.
- 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 typedefs in C.
- C++ comparison operators return **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 additional cast operators (**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 (RTTI), via **typeid** and **dynamic_cast**.
- C++ includes exception handling.
- C++ has `std::vector` as part of its standard library instead of variable-length arrays as in C.
- 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 exclusively.
- C++ supports object-oriented programming without extensions.
- C++ does not require use of macros and careful information-hiding and abstraction for code portability.

Java

This is a comparison of the Java programming language 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.

Java was created initially to support network computing on embedded systems. Java was designed to be extremely portable, secure, multi-threaded and distributed, 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
backwards compatible with C	backwards compatibility with previous versions
execution efficiency	developer productivity
trusts the programmer	restrains the programmer's abilities
arbitrary memory access possible	memory access only through objects
concise expression	explicit operation
can arbitrarily override types	type safety
procedural or object-oriented	object-oriented
operator overloading	meaning of operators immutable
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.
- `const` 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 supports 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 `protected`.
- 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 and labeled continue statements provide some structured goto-like functionality. In fact, Java enforces structured control flow, 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 low-level operating system components. Similarly, many C++ compilers support inline assembler. In Java, assembly code can still be accessed as libraries, through the Java Native Interface. 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 and pass-by-value. As in C, the programmer can simulate by-reference parameters with by-value parameters and indirection. In Java, all parameters are passed by value, but object (non-primitive) parameters are reference values, meaning indirection 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 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 that guarantees consistent results across platforms, though normally a more lenient mode of operation is used to allow optimal floating-point performance.
- In C++, pointers 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 pointers or functors). The equivalent mechanism in Java uses object or interface references.
- C++ features programmer-defined operator overloading. The only overloaded operators in Java are the "+" and "+=" operators, which concatenate strings as well as performing addition.
- Java features standard API support for reflection and w:dynamic loading 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 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 interfaces.
- 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.
- Java has both language and standard library support for multi-threading. The `synchronized` keyword in Java provides simple and secure mutex locks to support multi-threaded applications. While mutex lock mechanisms are available through libraries in C++, the lack of language semantics makes writing thread safe code more difficult and error prone.

Memory management

- Java requires automatic garbage collection. Memory management in C++ is usually done by hand, or through smart pointers. 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 arrays 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) 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. In Java compound types are always allocated on the heap and collected by the garbage collector (except in virtual machines that use escape analysis to convert heap allocations to stack allocations).
    - C++ has destructors, while Java has finalizers. 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 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 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 – a reference 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 is one that has no reachable references to it. An unreachable object cannot be destructed (deallocated), and results in a memory leak. By contrast, in Java an object will not be deallocated by the garbage collector *until* it becomes unreachable (by the user program). (Note: *weak references* 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

- C++ standard library only provides components that are relatively general purpose, such as strings, containers, and I/O streams. 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 with C, and C libraries (such as the APIs of most operating systems) are directly accessible from C++. In Java, the richer functionality of the standard library is that it provides cross-platform access to many features typically available in platform-specific libraries. Direct access from Java to native operating system and hardware functions requires the use of the Java Native Interface.

Runtime

- C++ is normally compiled directly to machine code which is then executed directly by the operating system. Java is normally compiled to byte-code which the Java virtual machine (JVM) then either interprets or JIT 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 overflows, page faults, and segmentation faults. The Standard Template Library, however, provides higher-level abstractions (like vector, list and map) to help avoid such errors. In Java, such errors either simply cannot occur or are detected by the JVM and reported to the application in the form of an exception.
- In Java, w:bounds checking 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. C++ uses a header file source code inclusion system for sharing declarations between source files. (See Comparison of imports and includes.)
- Templates and macros in C++, including those in the standard library, can result in duplication of similar code after compilation. Second, dynamic linking with standard libraries eliminates binding the libraries at compile time.
- C++ compilation features a textual preprocessing 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.Vector** respectively) which are resizable and store their size.
- 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 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 Level 6 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. For example:

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

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

While this may still be the case for embedded systems because of the requirement for a small footprint, advances in just in time (JIT) 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++. It's a bit pro-Java but very detailed.

C#

C# (pronounced "See Sharp") is a multi-purpose computer programming language suitable for all development needs. There is a WikiBook (http://en.wikibooks.org/wiki/C_sharp) that introduces C# language fundamentals and covers a variety of the base class libraries (BCL) provided by the Microsoft .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 an object, among other things. C# provides many additions to facilitate interaction with Microsoft's Windows, COM, and Visual Basic.

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** 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.

TODO
Should refer MS & MONO and Portable.NET http://getdotgnu.com/pnet, as well as the ECMA and ISO standards for C# and CLI

Managed C++ (C++/CLI)

Managed C++ is a shorthand notation for Managed Extensions for C++, which are part of the .NET framework from Microsoft. 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, which is an object-oriented machine language and was formerly known as MSIL

The Code

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). 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 with all of the proper commands, 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".

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.

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 is the strings of 1's and 0's that the computer can fully 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. 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 why will be discussed further in the Coding Style Section.

File Organization

Most operating systems require C++ files to be designated by a specific extension. The most common extension for a C++ file is ".cpp" for implementation code (the code that the programmer wrote to perform a given task), and ".h" for declaration or "header" files (though ".hpp" is becoming increasingly popular). Header files are a complicated concept that will be discussed with more detail later, but in general terms a header file is a special kind of source code that traditionally appears at the beginning of a complete file. There are other common extensions 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, ".hc", ".H", ".hxx", ".hpp" etc. Please note that file extensions don't include quotes; the quotes were added for clarity in this text. Regardless of what the specific form of the extension is, it serves one purpose: telling the Operating System, the IDE or the compiler that the text within the file is C++ source code, and not just random characters and so enable them to perform the necessary actions.

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.

TODO
Add reference to over .cpp .h, common rules to file naming and code distribution, live declaration vs implementation to the proper place and some parts should go into the optimization section probably were more should be said about it, remember to make reference on the #include section.

.cpp

.h

Object files

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 this 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 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.

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 of this book.

Statements

Most programming languages have the concept of a statement. A statement is a command that the programmer gives to the computer. It is also sometimes referred as an *expression*.

Example

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

A clear indicator that a line of code is a statement is its termination with an ending semicolon (;). Each statement performs an action. That statement and command will be examined in detail later on, for now

consider only, it as a verb ("cout") and the other details as information (what to print). In this case, the command "cout" means "show on the screen," (not "print on the printer").

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

If one were to write multiple statements, it is recommended that each statement be entered on a separate line and should end with a semicolon (;).

```
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 of the book.

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

The computer will perform each of these commands sequentially. It's 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 manual for the programming language you're using.

In the above case, the computer will look at the first statement, determine that it's 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!

Note that the quotation marks aren't 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!";
```

Statement Blocks

Also referred to Code Blocks (or in C++-speak, a *compound statement*), consist on one or more statements or commands that are contained between a pair of curly braces { }. Such a block of statement can be named or be provided a condition for execution. Below is how you'd place a series of statements in a block.

```
{
```

```
    int a = 10;
    int b = 20;
    int result = a + b;
}
```

A code block is 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.

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. A statement can be substituted by a statement block. 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 of the book.

Coding Style Conventions

As seen earlier, indentation and the use of white spaces or tabs are completely ignored by the compiler. However, a style guide or code convention goes beyond that to give programmers a fixed structure or *style* on how they should format their code, name their variables, place their comments or any other non language dependent structural decision that is used on the code. This can be very important, as you share a project with others. Agreeing to a set of rules enables a common set of coding standards and recommendations that will enable a greater understandings and transparency of the code base, providing a common ground for undocumented structures, making for easy debugging, and will increase code maintainability. These rules can be referred to as **Source Code Style**.

A list of different approaches can be found on the Reference Section. The most commonly used style for the C++ (and C) language is the Kernighan and Ritchie (K&R) style-guide. You should be warned that this is one of the first decisions as you take 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, so that you can adapt easily to changes or defend your approach. There is software that can help to format or *beautify* the code, but automation can have it's drawbacks.

Standardization is Important

It does not matter which particular coding style you pick. However, once a coding style 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.

Identifier Naming

To recall the definition for C++ Identifier:

> Definition:
>
> *Identifiers* 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++ 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.

This leaves a lot of freedom in naming. It is suggested that you also follow these rules:

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 isn't accessible anymore, it is hidden from external world) so unless you want to override an internal variable of a library, don't 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.

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. 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.

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's 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 don't 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*

Reduced use of keywords

Like for instance not using inline if the member function is implicitly inlined.

NOTE:
This can be defended both ways, more writing implies more work and possible errors but a clear statement of intention makes the code more readable and also reduces errors.

25 lines 80 columns

This is a commonly recommended but often inapplicable rule. Many people say it's an outdated rule, that it comes from prehistoric times when terminals could only display 25 lines 80 columns.

This rules signifies 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**. This recommended practice relates also to the *0 means success* convention for functions, that we will cover on the Functions Section of this book.

This practice will will save you precious time when you have to return to a project you haven't been working on for 6 months.

Whitespace and Indentation

Definition:

> 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 2 spaces.

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

```
// 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)

As we have seen early on the Statements Section, *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). 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)
   {
      // 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.

Definition:

- Single-line (informally, "C++-style") comments 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 (informally, "C-style") comments start with `/*` and end with `*/`

NOTE:
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 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 isn't in the comment */` at the end of the line which is not inside the comment; the comment ends at the first `*/` pair it finds, ignoring any interim `/*` pairs which might look to human readers like the start of a nested comment.

C++ Comments

C++ comments start with `//` and continue until the end of the line. or find the backslash \ character.

Example:

```
// 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.

With C comments

C-style comments (that start with `/*` and end with `*/`) can stop before the end of the line and can be used to "comment out" a small portion of code within a line in the program since there is no restriction to what is contained between the comment delimiters.

Example:

```
/* 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 become illegible when several #if 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:
One should avoid mixing comment solutions in release code but may do so to do a quick test/debug.
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.

If your comment lies into one line with code, use C++ style.

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.

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 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 sitting;
- 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.

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 community say that excessive commenting is indicative of code smell -- which is *not* to say that comments are bad, but that they are often a clue that code would benefit from refactoring. 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

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.

TODO
Add more here.

How ?

Documentation can be done within the source code itself through the use of comments as seen above. 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:

```
/*
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. 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 is a whole school of thought on how to approach this, but a very effective tool is Doxygen (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 man 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 pseudocode

You should refrain from using comments to do ASCII art or pseudocode (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.

Chapter Summary

1. Introducing C++
2. Programming languages
    1. Programming Paradigms - the versatility of C++ as a multi-paradigm language, concepts of Object-Oriented Programming (Objects and Classes, Inheritance, Polymorphism).
3. Comparisons - to other languages, relation to other computer science constructs and idioms.
    1. with C
    2. with Java
    3. with C#
    4. with Managed C++ (C++/CLI)
4. Code
    1. File Organization - implementation and declaration files (.cpp/.h).
    2. Statements - program control flow and statement blocks.
    3. Coding Style Conventions
    4. Documentation

Fundamentals

The Compiler

A **compiler** is a program that translates a computer program written in one computer language (the source code) into an equivalent program written in the computer's native machine language. This process of translation is called **compilation**.

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.

One of most actualized and compatible compilers is GCC. The next section will show how to get a copy and install it on Windows. You can easily find information on the GCC website on how to do it under another OS. GCC is a decent choice, and can be obtained for free. Many Open Source platforms include a recent GCC version. Version 4.0 or later gives fairly good conformance to the C++ standard. Various IDEs are available to support GCC. For Windows, Microsoft Visual Studio Express is currently available free of charge (but not free as in non-propriety) with a C++ compiler that can be used from the command line or from the supplied IDE (an IDE is generally a graphical environment which integrates functionality

like editing, compiling, linking, and usually a help system etc.).

NOTE:
In Appendix B:External References you will find references to other freely available compilers and even full IDEs you can use.

GCC

The GCC is a free set of compilers developed by the Free Software Foundation, with Richard Stallman as one of the main architects.

There are many different pre-compiled GCC compiler on the Internet, below shows you some popular choices with detailed steps for installation.

On Windows

Cygwin:

1. Go to http://www.cygwin.com 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-core" 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 related 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 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).
7. Type "gcc" at the Cygwin prompt and press "enter"; if "gcc: no input files" or something like it appears you have succeeded and now have the gcc 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, choose the version you want (eventually scrolling down), 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 handfull of clicks to let your first program run.

For DOS

DJGPP:

- Go to Delorie Software and click on *Zip Picker*and select the packages you need.
- Use unzip32 to inflate the files into the directory of your choice (ie. C:\DJGPP).

TODO

- Complete setup instructions for DJGPP.
- Add more examples of compilers with detail installation steps, such as MinGW.

For Linux

- For Redhat, get a gcc-c++ RPM, e.g. using Rpmfind and then install (as root) using `rpm -ivh gcc-c++-version-release.arch.rpm`
- For Fedora Core, install the GCC C++ compiler (as root) by using `yum install gcc-c++`
- For Mandrake, install the GCC C++ compiler (as root) by using `urpmi gcc-c++`
- For Debian, install the GCC C++ compiler (as root) by using `apt-get install g++`
- For Ubuntu, install the GCC C++ compiler by using `sudo apt-get install g++`
- 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.

Compilation

The *compilation* output of a compiler from translating or *compiling* a program is saved to a file called an **object file**. As we have seen before in the The Code Section of the book it consist on the transformation of source files into object files (some files may be created only with data that isn't part of the C++ language or resulting from the compilation of the same, this may depend on the compiler you use, you should check the documentation.

The instructions of this *compiled* program can then be run (executed) by the computer, if the object file is in an executable format, but often there is an additional steps that may be required to create an executable program, this are preprocessing and linking.

compile time

Defines the time and operations performed by a compiler (ie, *compile-time operations*) during a build (creation) of a program (executable or not).

The operations performed at compile time usually include *lexical analysis*, *syntax analysis*, various kinds of semantic analysis (eg, type checks and instantiation of template) and code generation.

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 (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. The concept of runtime will be introduced later.

TODO
Add run time concept, and mention it here (probably on Debugging)

lexical analysis

This happens before syntax analysis and converts the code into tokens, which are the parts of the code that the program will actually use, with 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. It uses whitespace to separate different tokens, and ignores the whitespace. To give an example

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

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 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.

TODO

Make this closer to what actually happens. This is a very simple and probably wrong example.

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 that 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't 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 inter, 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 didn't 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 don't 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 aren't 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.

ISO C++ (C++98) Keywords

- **and**
- **and_eq**
- **asm**
- **auto**
- **bitand**
- **bitor**
- **bool**
- **break**
- **case**
- **catch**
- **char**
- **class**
- **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**
- **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-std.org/jtc1/sc22/wg21/docs/papers/2006/n2105.html 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`**

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

TODO
It would be nice to list those C reserved identifiers

Compiler keywords

A limited set of 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.

The C++ Standard calls this keywords, specifiers.

auto

The `auto` keyword used to have a different behavior, but in C++0x it will allow one to omit the type of a variable and let the compiler decide. This is particularly useful for generic programming in which the return type of a function may depend on the type of its arguments. Thus, rather than this:

```
int x = 42;
std::vector<double> numbers;
numbers.push_back(1.0);
numbers.push_back(2.0);
for(std::vector<double>::iterator i = numbers.begin();
      i != numbers.end(); ++i) {
   cout << *i << " ";
}
```

we could write this:

```
auto x = 42; // We can use auto on base types...
std::vector<double> numbers;
numbers.push_back(1.0);
numbers.push_back(2.0);
// But auto is most useful for complicated types.
for(auto i = numbers.begin(); i != numbers.end(); ++i) {
   cout << *i << " ";
}
```

Note: This functionality is not yet available.

inline

A function declaration with an inline keyword declares an inline function. The **inline** keyword is used to suggest to the compiler that a particular function be subjected to in-line expansion; 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 back again to execute a subroutine, as is done in naive implementations of subroutines.

Example:

```
inline swap( int& a, int& b) { int const tmp(b); b=a; a=tmp; }
```

Marking a function as **inline** (possibly implicitly, by defining a member function inside a class/struct definition) is a (non-binding) request to the compiler to consider inlining the function, i.e., expanding its code at the call site; it is legal, but redundant, to add the `inline` keyword in that context, and good style 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-inline 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-inline 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++ and GCC 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.

Excessive use of **inline** functions can greatly increase coupling/dependencies and compilation time, as well as making header files less useful as documentation of interfaces.

extern

The extern keyword tells the compiler that a variable is declared in another source module. The linker then finds this actual declaration and sets up the extern variable to point to the correct location. 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.

Storage Class Specifiers

- `register` - A hint to the compiler that the specified variable will be heavily used; therefore the compiler should consider allocating a CPU register to the variable. The compiler may ignore this hint.
- `static` - Retains a memory location for all instances of the program or class.

Compile Speed

Most problems one has with a slow compilation are due to:

- Hardware

  Resources (Slow CPU, low memory and even a slow HD can have an influence)

- Software

  The compiler itself (new is probably better), the design used on the program (structure of object dependencies, includes)

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.

Use pre-compiled headers and external header guards.

The Preprocessor

The preprocessor is either a separate program invoked by the compiler or part of the compiler itself, which performs intermediate operations that modifies the original source code and internal compiler options before the compiler tries to compile the resulting source code.

The instructions that the preprocessor parses 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 (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" (http://en.wikibooks.org/wiki/Compiler_Construction)

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

Inclusion of Header Files (#include)

The **#include** directive allows a programmer to include contents of one file inside another one. 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 repeatedly type out all the information.

C++ generally requires you to *declare* what will be used before using it. So, files called **headers** usually include declarations of what will be used in order for the compiler to successfully compile source code. The **standard library** (a repository of code that is available alongside every standard-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 to make use of features/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 causes the inclusion of the contents of 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 using angle brackets (<>), the preprocessor is instructed to search for the file to include in a compiler-dependent location. When you use quotation marks (" "), 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. It is common for this form to include searching 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.

NOTE:
Compilers are allowed to make an exception in the case of standard library as to whether a header file by a given name actually exists or just has the same effect as if the header file did exist. 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

- algorithm
- bitset
- complex
- deque
- exception
- fstream
- functional
- iomanip
- ios
- iosfwd
- iostream
- istream
- iterator
- limits
- list
- locale
- map
- memory
- new
- numeric

and the

Standard C Library

- cassert
- cctype
- cerrno
- cfloat
- ciso646
- climits
- clocale
- cmath
- csetjmp
- csignal
- cstdarg
- cstddef
- cstdio
- cstdlib
- cstring
- ctime
- cwchar
- cwctype

Everything inside C++'s standard library is kept in the `std::` namespace. Old compilers may include headers before C++ was standardized, named `<X.h>` and `<cX.h>`, in addition to or instead of the standard headers. Often these headers have non-templatized classes and pollute the global namespace. Some have the SGI STL on which much of the standard template library is based.

Non-standard but somewhat common C++ libraries

- stdiostream.h[1]
- stream.h[2]
- strstream.h[3]

1. ↑ Streams based on FILE* from stdio.h.
2. ↑ Precursor to iostream. Old stream library mostly included for backwards compatibility even with old compilers.

3. ↥ Uses **char*** whereas sstream uses string. Prefer the standard library sstream.

#pragma

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

Pragmas are used within the source program.

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

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

For instance one of the most implemented preprocessor 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:
Other methods exist to do this action that is commonly referred to as using **include guards**.

NOTE:
In 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 don't 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/) the project maintains a complete list of macros that are compiler and OS agnostic.

#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

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 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`.

\ (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 preprocess 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 won't arise often.

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

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

That's 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. Don't 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 ) ) {
...
}
```

It's generally a good idea to use extra parentheses when using complex macros. 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 won't compile or create 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++) );
```

- 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")

Example:

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 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);
```

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 isn't a problem: just writing two adjacent string literals is enough to make the preprocessor concatenate them.

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's 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 won't 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 conditionN
   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. 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 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 read the line `#include "foo.hpp"`, the content of **foo.hpp** will be expanded. Since this is the first time which **foo.hpp** is read, (assuming that there is no existing declaration of macro **FOO_HPP**,) **FOO_HPP** wont be declared due to its inexistence, and so the code will be added in 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 target - avoid the same file being included for 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

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.

Linker

The **linker** is a program that is responsible for linking and resolving linkage issues, such as the use of symbols or identifiers which are defined in one translation unit and are needed from other translation units, this information is created by the compiler. Symbols or identifiers which are needed outside a single translation unit must 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 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.

C++ programs can be compiled and linked with programs written in other languages, such as C, Fortran, and Pascal. When programs have two or more source programs written in different languages, you should do the following:

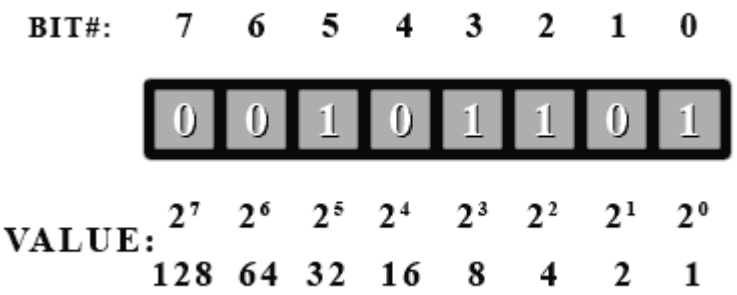
- Compile each program module separately with the appropriate compiler.
- Link them together in a separate step.

Internal storage of data types

Bits and Bytes

The byte is the smallest individual piece of data that we can access or modify on a computer. The computer only works on bytes or groups of bytes, never on bits. If you want to modify individual bits, you have to use binary operations on the whole byte that tell the computer how to modify individual bits, but the operation is still done on whole bytes. Before getting too far ahead of ourselves, we'll look at the internal representation of a byte.

Here's a look at a byte as the computer stores it.



There is actually quite a lot of information here. A byte (usually) contains 8 bits. A bit can only have a value of 0 or 1. The bit number is used to label each bit in the byte (so that we can tell which bit we are talking about). You may be wondering why the bits are labeled from 7 to 0 instead of 0 to 7 or even 1 to 8. The reason 0 is used is because computers always start counting at 0. Technically, we COULD start counting at 1, but this would go against the counting nature of the computer. It is simply more convenient to use 0 for computers as we shall see. Now as to why we numbered them in descending order. In decimal numbers (normal base 10), we put the more significant digits to the left. Example: 254. 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. Counting in binary and in decimal is done in exactly the same manner, except that in binary, instead of counting from 0 to 9, we only count from 0 to 1. If we want to count higher than 1, then we need 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 we as humans are used to counting with 10 digits. In binary, there are only 2 digits, but counting is done by the exact same principles as counting in decimal.

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 reperesent numbers. However, we won't look into that right now.

In decimal, each digit represents multiple of a power of 10. Let's take another look at 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)

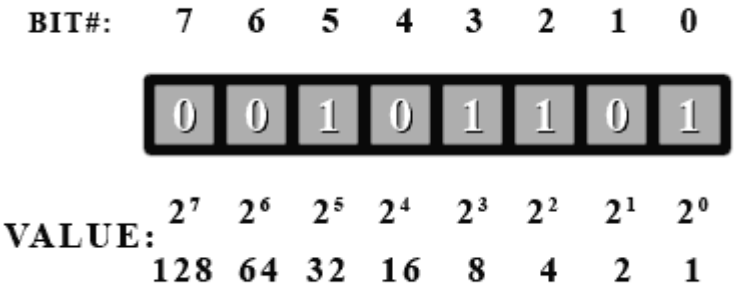
All this is elementary. The key point to recognize is that as we move from right to left in the number, the significance of the digits increases by a multiple of 10. This should be obvious when we look at the following equation:

$$(2 \times 10^2) + (5 \times 10^1) + (4 \times 10^0) = 254$$

Do you see any similarities between this and the diagram above? 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 you should find very much like the one above:

$$(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 8 + 4 + 0 + 1 = 13$$

So, 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). The bases that these numbers are in are shown in subscript to the right of the number.

$00_2 = 0_{10}$

$01_2 = 1_{10}$

$10_2 = 2_{10}$

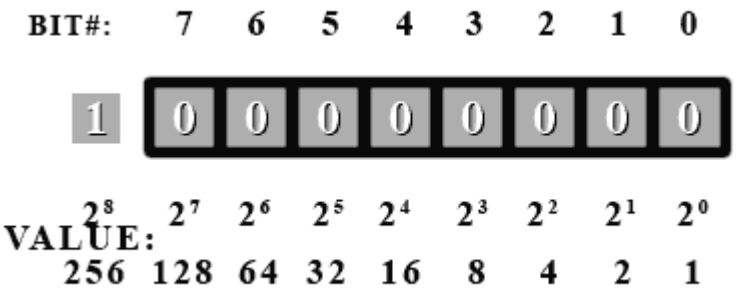
$11_2 = 3_{10}$

Data and Variables

When programming in C++, we need a way to store data that can be manipulated by our program. Data comes in a variety of formats, so the compiler needs a way to differentiate between the different types. Right now, we'll concentrate on using bytes. The type name for a byte in C++ is 'char'. It's called char because a byte is often used to represent characters. We won't go into that right now. We only want to use it for its numerical representation.

Let's write a program that will print each value that a byte can hold. How do we do that? We could write a

loop that goes from 0 to 255. We'll set our byte to 0 and add one to it every time through the loop. As a side note, do you know what would happen if you added 1 to 255? No combination will represent 256 unless we add more bits. If you look at the diagram above, you will see that 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? It actually goes into the carry bit. The carry bit, you say? The processor of the computer has an internal bit used exclusively for carry operations such as this. So if you add 1 to 255 stored in a byte, you'd get 0 with the carry bit set in the CPU. Of course, being a C++ programmer, you never get to use this bit directly. You'll need to learn assembler if you want to do that, but that's a whole other ball game.

In our program, we can start off with a value of 0 and wait until it becomes 0 again before exiting. This will make sure we go through every value a byte can hold.

Inside your main() function, write the following. Don't worry about the loop just yet. We are more concerned with the output right now.

```
char b=0;

do
{
   cout << (int)b << " ";
   b++;
   if ((b&15)==0) cout << endl;
} while(b!=0);
```

b is our byte and we initialize it to 0. Inside the loop we print its value. We cast it to an int so that a number is printed instead of a character. Don't worry about casting or int's right now. The next line increments the value in our byte *b*. Then we print a new line (carriage return/endl) after every 16 numbers. We do this so that we can see all 256 values on the screen at once.

If you were to run this program, you would notice something strange. After 127, we got -128! Negative numbers! Where did these come from? Well, it just so happens that the compiler needs to be told if we're using numbers that can be negative or number that can only be positive. These are called signed and unsigned numbers respectively. By default, the compiler assumes we want to use signed numbers unless we explicitly tell it otherwise. To fix our little problem, add "unsigned" in front of the declaration for b so that it reads: "unsigned char b=0;" (without the quotes). Problem solved!

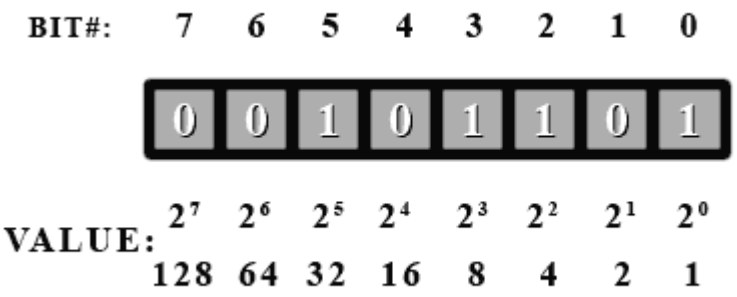
Two's Complement

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.

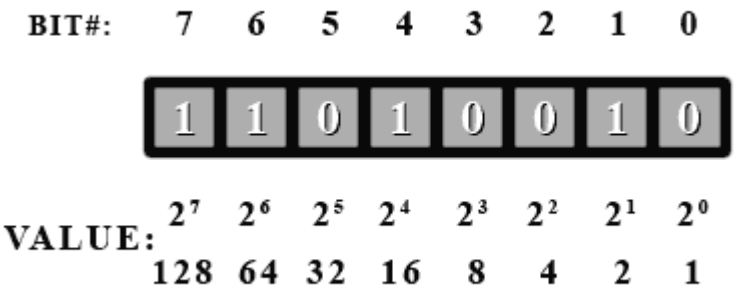
To convert a positive number into it's negative two's complement format, you begin by flipping all the bits in the number (1's become 0's and 0's become 1's) and then add 1. (This also works to turn a negative

number back into a positive number Ex: -34 into 34 or vice-versa).

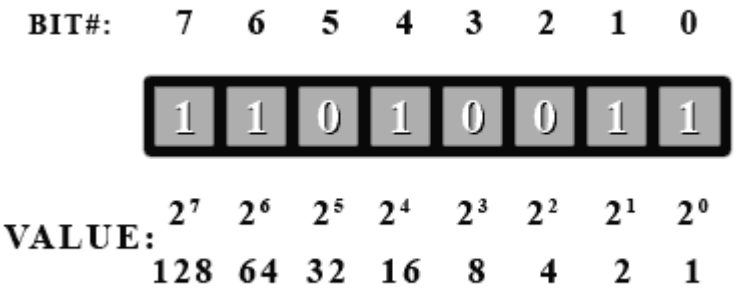
Let's try to convert our number 45.



First, we flip all the bits...



And add 1.



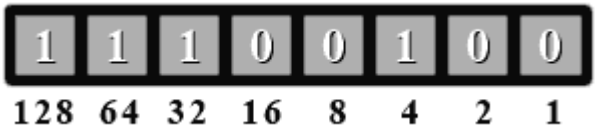
Now if we add up the values for all the one bits, we get... 128+64+16+2+1=211? What happened here? Well, this number actually is 211. It all depends on how you interpret it. If you decide this number is unsigned, then it's value is 211. But if you decide it's signed, then it's value is -45. It is completely 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?

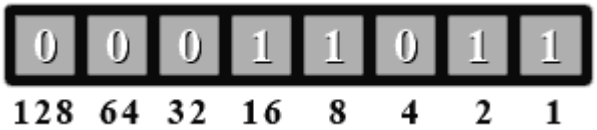


Since this is an unsigned number, no special handling is needed. Just add up all the values where there's a 1 bit. 128+64+32+4=228. So this binary number is 228 in decimal.

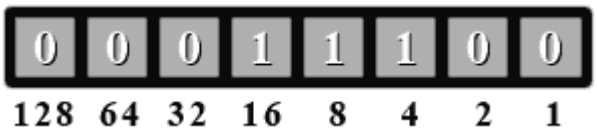
Now treat the number above as a signed byte. What is its value in decimal?

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 it's value as a positive number (then we'll add the negative sign afterwards).

Ok, so let's flip all the bits...



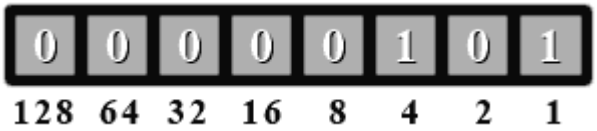
And add 1. This is a little trickier since a carry propagates to the third bit. For bit#0, we do 1+1 = 10 in binary. So we have a 0 in bit#0. Now we have to add the carry to the second bit (bit#1). 1+1=10. bit#1 is 0 and again we carry a 1 over to the 3rd bit (bit#2). 0+1 = 1 and we're done the conversion.



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.



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.

Endian

Now that we've seen many ways to use a byte, it is time to look at ways to represent numbers larger than 255. 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 us to represent numbers up to 65535 (unsigned). And 32 bits allows us to represent numbers above 4 billion. We already saw the type for a byte. It is called a 'char'.

Here are a few basic primitive types:

1. char (1 byte (by definition), max unsigned value: at least 255)

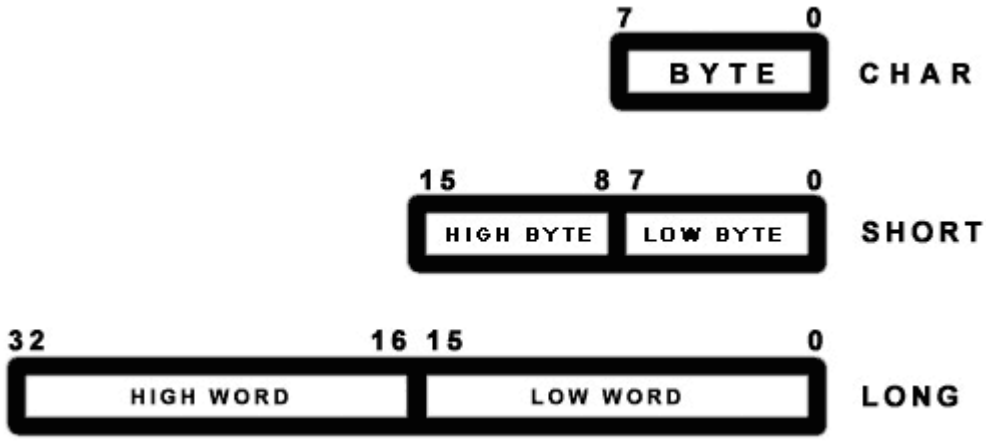
2. short int (at least 16 bits, max unsigned value: at least 65535)

3. long int (at least 32 bits, max unsigned value: at least 4294967295)

4. float (typically 4 bytes, floating point)

5. double (typically 8 bytes, floating point)

For 'short int' and 'long int', you can leave out the 'int' because 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 of the topics explained above also apply to short int's and long int's. 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).

Let's look at a (16-bit) short. You may think that in memory the byte for bits 15 to 8 would be followed by the byte for bits 7 to 0 (because bits 15 to 8 appears first). 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 work this way. The Amiga and old Macintoshes use the M68000 and they indeed 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.

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 in 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.

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:

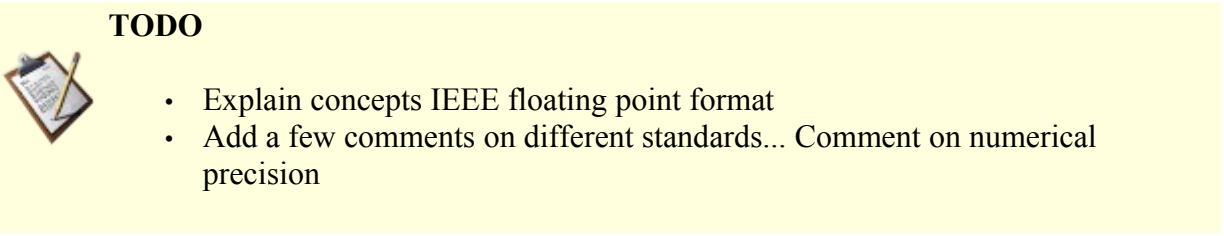
$$f = s \times M \times 2^{e-E}$$

S, e and M are concatenated one after the other in 32-bit words to create a float number and in 64-bit words to create a double one. For the 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 53 for the mantissa and the exponent offset is E=1023. Note that, the (non-zero) digit before the decimal point in the normalized binary representation has to be 1, so it is not stored as part of the mantissa.

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.



TODO

- Explain concepts IEEE floating point format
- Add a few comments on different standards... Comment on numerical precision

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. Typically a variable is bound to a particular address in computer memory 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 stored in that particular memory location. If the size and location of a variable is unknown beforehand, then where an object instance can be found is used as the value of the variable instead and the size of the variable is determined by the size needed to hold the location value. This is called referencing.

Variables reside in a specific scope. The scope of a variable 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. This becomes important later as the constructor of variables are called when entering scope and the destructor of variables are called when leaving scope. 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 block scope.

Types

Just as there are different types of values (integer, character, etc.), there are different types of variables. A variable can refer to simple values like integers and strings called a *primitive type* or to a set of values called a *composite type* that are made up of primitive types and other composite types. 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 cannot be divided by an integer in the usual sense. As discussed below, strong typing offers more safety, but it does not necessarily guarantee complete safety (see type-safety 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 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 types; 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.

Table of Data Types

Type	Size in Bits	Comments	Alternate Names
		Primitive Types	
char	≥ 8	• **sizeof** gives the size in units of **char**s. These "C++ bytes" 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].	—
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]. • 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**
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].	**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].	**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.	—
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	• **unsigned** cannot be specified.	—

	double		
		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 is to use **struct** only for POD types. • Said to be a *compound type*.	—
union	≥ size of the largest member	• Default access modifier for 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	• **typedef** has syntax similar to a storage class like **static**, **register** or **extern**.	—
template	≥ size of **char**	—	—
		Derived Types[4]	
type`&` (reference)	≥ size of **char**	• References (unless optimzed 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.	—
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.	—

Table of Data Types Footnotes

[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

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).

standard types

C++ has five basic primitive types called **standard types**, specified by particular keywords, that store a single value.

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 `!=`).

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;
}
```

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;
}
```

Type 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.

```
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 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 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 == 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` specifier 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. 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.

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 use `unsigned char`.

Enumeration Types

An **enum** or enumerated type is a type that describes a set of named values. Every enumerated type is stored as an integral type. While it is possible to assign a constant integer value to a named value of an enumerated type, most often implicit integer values will be used, with the first named value having an integer value of 0 and all others in turn having an integer value one more than its predecessor.

For example:

```
enum colour {Red, Green, Blue};
```

defines colour as a set of named constants called **Red**, **Green** and **Blue** with values of *0*, *1* and *2*. Enumerations are useful for when you have set of related values which can help to make code more readable by removing magic numbers. Consider the following:

```
if (yourColour == Red)
{
   std::cout << "You chose red" << std::endl;
}
```

to the slightly less meaningful

```
if (yourColour == 0)
{
   std::cout << "You chose red" << std::endl;
}
```

In the first example using the enum makes the meaning clearer, whereas in the second example one may wonder what the 0 means. By consistently using enumerated types in a program rather than hard coding values the code becomes easier to understand and reduces the chance of accidental error such as the quandary, *"Was it a 0 or a 1 that should be tested for the colour red?"*

I can also count with enums, such as:

```
enum month
{ JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER };

for (int monthcount = JANUARY; monthcount <= DECEMBER; monthcount++)
{
    std::cout << monthcount << std::endl;
}
```

Enumerations do not necessarily need to begin at 0, and although each successive variable is always increased by one, this can be overridden by assigning a value.

```
enum colour {Red=2, Green, Blue=6, Orange};
```

In the above example **Red** is **2**, **Green** is **3**, **Blue** is **6** and **Orange** is **7**.

User Input

In the previous program two numbers are added together which are specified as part of the program. If we want to find the sum of any other pair of numbers using the above program, we would have to edit the program, change the numbers, save and recompile. In other words, the numbers are *hard-coded* into the program and any change requires recompilation.

It is always better to write programs to be more flexible, in that they do not assume any values but allow the user to specify them. One way to allow the user to specify values is to prompt the user for the values and get them, like so:

```
Enter number 1: 230
Enter number 2: -35
The sum of 230 and -35 is 195.
```

Text that is *italicized* is typed by the user and the **bold** text is output by the program.

Derived Types

TODO
Complete... ref to:
C++ Programming/Operators/Pointers
C++ Programming/Operators/Arrays

Scope

As with any other type of language, context (i.e. what is the background of a given action or statement) has a high impact on its validity or value. The same happens in a programming language. For instance, variables have a finite lifetime when your program executes. We will see that in a program we have various constructs, may they be objects, variables or any other such, and they come into existence from the point were 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.

The scope of a object or variable is simply that part a program in which the variable name is valid (i.e. it exists). For example, in the following fragment of code, the variable 'i' is in scope only in the lines between the appropriate comments:

```
{
   char ch;
   ch = 'X';
   int i; /*'i' is now in scope */
   i = 5;
   ch = 'A';
   i = i + 1;
   cout << i << ch;
}/* 'i' is now no longer in scope */
```

The concept of scope is straightforward unless procedures are included in the mix; then it becomes more difficult to follow:

```
// Confusing Scope Program
#include <iostream>

using namespace std;

int i = 5;           /* The first version of the variable 'i' is now in scope */
void p(){
   int i = -1;        /* A ''new'' variable, also called 'i' has come into existence. The 'i' declared above is now out
of scope,*/
   i = i + 1;
   cout << i << ' ';  /* so this ''local'' 'i' will print out the value 0.*/
}                    /* The newest variable 'i' is now out of scope. The first variable, also called 'i', is in scope
again now.*/
main(){
   cout << i << ' ';  /* The first variable 'i' is still in scope here so a ''5'' will be output here.*/
   char ch;
   int i = 6;         /* A ''new'' variable, also called 'i' has come into existence. The first variable 'i' is now out
of scope again,*/
   i = i + 1;
   p();
   cout << i << endl; /* so this line will print out a ''7''.*/
}                    /* End of program: all variables are, of course, now out of scope.*/
```

The first variable 'i' is put out of scope in two separate sections. Thus the repeated statement *cout << i << ' ';* means something very different each time it is written. The 'i' referred to is a different location in memory on each occasion. This is what was meant by `context' in the introduction: the context or background in which the statement is placed of is different each time, so the statement does something different in each place.

It is always an error to declare the same variable name twice within the same level of scope. Thus the second-last line of the program would cause an error if it were uncommented.

The purpose of Scope

There are important things to note about the above example. That program is only an example program and unusually convoluted; it is useful to demonstrate the idea of scope and little else. So while it illustrates the concept of scope, it fails to illustrate usefully the purpose of scope.

Some variables are required to store information for an entire program, while other variables are short-term variables which are brought into existence momentarily for a single small purpose and then disposed of, by going out of scope. In the following program, an array of numbers is read in and then a procedure is called that computes the average of the numbers in the array. Within the procedure, in order to move through the array and select the elements in the array in turn, a variable 'i' is created for that one purpose. Contrast the two kinds of variable: the array itself, which is in scope throughout the entire program, and the variable 'i' which is in scope only for a small section of the code, to do its own little job.

```
// Program Average
#include <iostream>

using namespace std;

float a[20];                    /* a is now in scope.*/
int length;                     /* length is now in scope.*/
```

```
float average(){
   float result = 0.0;             /* result is now in scope.*/

   for(int i = 0; i < length; i++){ /* i is now in scope.*/
     result += a[i];
   }

   return result/length;
}                                  /* result and i are now out of scope.*/

int main(){
  length = 0;
  while(cin >> a[length++]){
  };
  length--;
  float av = average();            /* av is now in scope.*/
  cout << av << endl;
}                                  /* All variables now out of scope.*/
```

Scope and control structures

Within a single procedure it is possible to begin new a level of scoping. In fact it occurs every time a left brace `{' is written and it ends where its matching right brace '}' is written. Thus layers of scope to any depth can be build up as can be seen in the following program. The following program has four levels of scoping. The innermost level is said to be *enclosed* by the levels around it, thus we talk about inner levels and enclosing levels of scoping. Again this program is for illustration purposes only and does nothing of real value.

```
// Complicated Scope Program
#include <iostream>

using namespace std;  /* outermost level of scope starts here */

int i;

main(){               /* next level of scope starts here */
   int i;
   i = 5;

   {                   /* next level of scope starts here */
      int j,i;
      j = 1;
      i = 0;

      {                 /* innermost level of scope of this program starts here */
         int k, i;
         i = -1;
         j = 6;
         k = 2;
      }                 /* innermost level of scope of this program ends here */

      cout << j << ' ';
   }                   /* next level of scope ends here */

   cout << i << endl;
}                     /* next and outermost levels of scope end here */
```

The output if this program is *6 5*. To understand why, we look first at the simpler situation of 'i', and we see why a five is printed for it, and then next at the more complicated situation with 'j' and why a six is printed for it.

A new variable 'i' is created at each new level of scope. Thus only the first assignment to 'i', where 'i' is assigned the value of five, affects this particular variable 'i': the variable that is printed. That variable does not alter its value after that first assignment and so the final statement prints a five. The other assignments to 'i' are irrelevant to the final print statement.

In contrast, the variable 'j' is created only once, hence the assignment where `j' is assigned the value of six, alters this only existing variable called 'j' even though the variable is declared at an enclosing level of scope. If a program has a scope level inside another, and no variable of the right name is declared at this inner level, then the computer `looks outwards' to the next enclosing level of scope. If there is no variable of that name declared there either, then it will look out further and again further outwards until the variable's declaration is found. (Of course, if a declaration for the variable is *never* found then the compiler indicates an error, stating that the variable is undeclared and so the program is not compiled.)

Scope using other control structures

Above we stated that a new level of scope started with each left brace `{'. However, it is not common to use just a naked left brace: usually the left brace is associated with an *if* statement or a *while* statement or some such. We add these to the above program to make the more usual (but still useless) example program following. The program does compile and it prints a *5* when it runs, but does little of value as a program.

```
// Complicated Scope Program, variation 1
#include <iostream>

using namespace std;  /* outermost level of scope starts here */

int i;

main(){               /* next level of scope starts here */
   int i;
   i = 5;

   while(i != 5) {     /* next level of scope starts here */
      int j,i;
      j = 1;
      i = 0;
      switch (i) {      /* next level of scope starts here */
         int i;

         case 1:
            if (i != 4) { /* innermost level of scope of this program starts here */
               int k, i;
               i = -1;
               j = 6;
               k = 2;
            }             /* innermost level of scope of this program ends here */
            break;

         case 2:
            j = 5;
            break;
      }                 /* next level of scope ends here */

      cout << j << ' ';
   }                   /* next level of scope ends here */

   cout << i << endl;
}                     /* next and outermost levels of scope end here */
```

We added an extra level of scoping at the *switch* statement because this statement demands a left brace, and it shows that even at that point a new level of scope is opened (because we were able to declare yet another variable called `i' without getting an error). As you can see, the various control structures (i.e. the *while, if* and *switch* statements) all end at their own matching right brace and hence at the same point as the scope ends. It is fair to say that each of these control statements operates over an area of scope. But remember, scope is a concept about variable names and the area in which they are defined, not control structures.

It is not essential that either the *if* or the *while* statements have an opening brace following, and *program*

average above shows an example of this with its *while* statement. In this situation, no new level of scope is begun. It is the left brace that opens the new level of scope, not the *if* or *while* statement itself. The *switch* statement demands a left brace at that point, but notice that the `i' switch variable is at the old level of scope. The new level of scope comes into existence immediately following the left brace as always.

For all practical purposes, the *for* loop control structure can be seen to have scoping that operates in the same manner. However it is permissible to declare the *for* loop variable within the *for* statement itself, as can be seen in line eight of *program average* above. This is good programming practice as it makes the structure of the program clearer.

With all the programs in this section the statements start further and further to the right as the level of scope deepens. This is referred to as *indentation* and is a very important feature of program presentation. It makes the scope level explicit at all times and also makes it clear where a statement ends. For example, in the above program the line *cout << j << ' ';* is within the *while* loop whereas *cout << i << endl;* is outside that loop. The loop and the scope level end at the same point: the right brace between these two statements signals the end of both of these.

The scoping of the *for* control statement in detail

This subsection is probably more confusing than useful, but it is offered for completeness; feel free to skip it.

The *for* control statement has an unusual scoping, in that the left *round* bracket also starts its own level of scope. Thus the following program is legal:

```
// Complicated Scope Program, variation 2
#include <iostream>

using namespace std; /* outermost level of scope starts here */

int i;

main(){              /* next level of scope starts here */
   int i;
   i = 5;

   for(               /* next level of scope starts here */
         int i = 1;
         i<10 and cout << i << ' ';
         ++i )
   {                  /* next level of scope starts here */
      int i = -1;
      cout << i << ' ';
   }                  /* two levels of scope end here*/

   cout << i << endl;
}                    /* next and outermost levels of scope end here */
```

and it gives the output:

```
 1 -1 2 -1 3 -1 4 -1 5 -1 6 -1 7 -1 8 -1 9 -1 5
```

This special feature of the *for* statement is not shared by, for example, the *while statement*. It is a syntax error to attempt to declare a variable within the *while* statement, so *while(int i < 22)i++;* gives a syntax error.

This special scope level enables one to declare a *for* loop variable within the *for* loop itself (like the variable 'i' in the above program), instead of having to declare it in the enclosing scope level, creating a neater program. But it is a little peculiar.

The above program does show one interesting feature: in order to check if a new level of scope has been

opened, it is only necessary to attempt to declare an existing variable again at that level, as was done with the variable `i' in the above example program, which has a new variable `i' declared at every level possible.

The above program also illustrates one further very important point: there is a saying in computer programming that is it possible to write bad code in any language. The above program is quite unclear in its operation and a good example of bad coding: it is good code to illustrate a point, but bad code to read. All code should be written to make the program as clear as possible as discussed elsewhere under the topic of program *style*.

Scope and lifetime in C++

The scope of a variable should be contrasted with its *lifetime*. In the program above called `Confusing Scope Program' the first variable `i' goes out of scope for a time but it remains in existence, thus its lifetime is continuing while it is out of scope. In older programming languages, it is difficult to contrive examples where scope and lifetime are different - in general in these older languages the two are the same, so lifetime equals scope. Not only is it difficult, it is not all that useful in the general case when it does occur. In recently created computer languages like C++ however, the idea of having variables that are out of scope but still alive is heavily exploited and creates the principle distinguishing feature of C++: the C++ *class*. This is the above program re-written with a class:

```
// Program Average rewritten using a class
#include <iostream>

using namespace std;

class StatisticsPackage{
private:
   float aa[20];                      /* aa scope start*/
   int length;                        /* length scope start*/
public:
   float average(){
      float result = 0.0;             /* result scope start*/

      for(int i = 0; i < length; ++i) /* i scope start*/
         result += aa[i];

   return result/length;
   }                                  /* result and i scope end*/

   void get_data(){
      length = 0;
      while(cin >> aa[length++]);
      --length;
   }
};                                    /* aa and length scope end*/

main(){
   StatisticsPackage sp;              /* aa and length lifetimes start */
   sp.get_data();
   float av = sp.average();           /* av scope start*/
   cout << av << endl;
}                                     /* av scope end*/
```

In this version of the program, the variables `length' and `aa' are alive after the class `sp' comes into existence. However, their scope has been limited: they are alive but out of scope, storing the information but not being directly available in the main program. Keeping variables out of scope in this manner is very helpful in debugging a program, by narrowing the number of lines in which the variable in question can possibly change its value.

TODO
evolve this simple introduction and add references to the further insight and practical usefulness

in sections like class space and flow control structures.

Namespace

The keyword as used on the several code examples examined before, this section will give an explanation to the line *using namespace std;*.

In many programming languages, a namespace is a context for identifiers. 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, don't use it in a way to make code overcomplicated for you and others

A namespace is defined with a namespace block.

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

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.

```
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 anonymous 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 anonymous namespace in an ordinary namespace, and vice versa, you can also nest two anonymous 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;
}

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';
}
```

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;
}
```

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 namespaces 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.

Operators

Operators are special symbols that are used to represent simple computations, this is significative importance in programming, since it serves to define the interaction of data in a useful way.

Computers are mathematical devices, but compilers and interpreters require a full syntactic theory of all operations in order to parse formulas involving any combinations correctly. In particular they depend on operator precedence rules, on order of operations, 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 (for functions).

C++ like all programming languages uses a set of operators, they are subdivided into several groups:

- arithmetic operators (like addition and multiplication).
- boolean operators.
- string operators (used to manipulate strings of text).
- 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 they are common mathematical symbols. For example, the operator for adding two integers is +. C++ allows the re-definition of some operators (operator overloading) and this be covered later on.

The following are all legal expressions whose meaning is more or less obvious:

- 1+1
- hour-1
- hour*60 + minute
- minute/60

take the line:

```
sum = a + b;
```

it uses the + operator to add the values stored in the locations `a` and `b` and the assignment operator (=) to store the result in the location `sum`. `a` and `b` are said to be the *operands* of +. The combination `a + b` is called an *expression*, specifically an *arithmetic expression* since + is an *arithmetic operator*. Similarly, = and its operands, `sum` and `a + b` together form the assignment expression `sum = a + b` (Note that the semicolon is not part of the expression). Other arithmetic operations that can be performed on integers (also common in many other languages) include:

- Subtraction, using the − operator
- Multiplication, using the `*` operator
- Division, using the `/` operator
- Remainder, using the `%` operator

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.

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.

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`	
Function Call, Member Access, Post-Increment/Decrement Operators, RTTI and C++ Casts			Left to right
`()`	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`	
`->`	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*	**`typeid`** `(std::cout)`	
`typeid``()`	run time type identification operator	**`typeid`** `(std::iostream)`	

	for a type	
static_cast<>()	new C++ style cast operator *for compile-time type conversion*	**static_cast**<**float**> (i)
dynamic_cast<>()	new C++ style cast operator *for type conversion using RTTI*	**dynamic_cast**<std::istream> (stream)
const_cast<>()	new C++ style cast operator *for removing the* **const** *qualifier from a pointer or a reference*	**const_cast**<**char***> ("Hello, World!")
reinterpret_cast<>()	new C++ style cast operator *for interpreting raw data as being of some other type*	**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	Right to left
!	logical not operator	!eof_reached
~	bitwise not operator	~mask
+ -	unary plus/minus operators	-num
++ --	pre-increment/decrement operators	++num
&	address-of operator	&data
*	indirection operator	*ptr
new	new operator *for single objects*	**new int** **new** std::string (5, '*')
new[]	new operator	**new int** [100]

	for dynamically allocated arrays		
new()	new operator with arguments *for single objects*	`new (raw_mem) int` `new (arg1, arg2) myobj (5, '*')`	
new()[]	new operator with arguments *for dynamically allocated arrays*	`new (arg1, arg2) int [100]`	
delete	delete operator *for pointers to single objects*	`delete ptr`	
delete[]	delete operator *for dynamically allocated arrays*	`delete[] arr`	
sizeof	sizeof operator *for expressions*	`sizeof 123`	
sizeof()	sizeof operator *for types*	`sizeof (int)`	
(*type*)	traditional/C-style cast operator *(deprecated)*	`(float)i`	
	Member Pointer Operators		
.*	member pointer access operator *for an object of class/union type or a referemce 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		Left to right

`+ -`	addition and subtraction operators	`end - start + 1`	
	Bitwise Shift Operators		
`<<`	left shift operator	`bits << shift_len`	Left to right
`>>`	right shift operator	`bits >> shift_len`	
	Relational Inequality Operators		
`< > <= >=`	less-than, greater-than, less-than or equal-to, greater-than or equal-to operators	`i < num_elements`	Left to right
	Relational Equality Operators		
`== !=`	equal-to, not-equal-to	`choice != 'n'`	Left to right
	Bitwise And Operator		
`&`		`bits & clear_mask_complement`	Left to right
	Bitwise Xor Operator		
`^`		`bits ^ invert_mask`	Left to right
	Bitwise Or Operator		
`|`		`bits | set_mask`	Left to right
	Logical And Operator		Left to right

`&&`		`arr != 0 && arr->len != 0`	
	Logical Or Operator		
`||`		`arr == 0 || arr->len == 0`	Left to right
	Conditional Operator		
`?:`		`size >= 0 ? size : 0`	Right to left
	Assignment Operators		
`=`	assignment operator	`i = 0`	
`+= -= *= /= %= &= |= ^= <<= >>=`	shorthand assignment operators *(`foo` op= `bar` represents `foo = foo` op `bar`)*	`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

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.

Chaining Insertion Operators

```
std::cout << "The sum of " << a << " and " << b << " is " << sum << "\n";
```

The 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.

Precedence (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.

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; operator overloading and reference types complicate the picture.)

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.

Assignment

The most basic assignment operator is the "=" operator. It assigns one variable to have the value of another. For instance, the statment `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 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 { /* ... */ };
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 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.

Arithmetic Operators

```
sum = a + b;
```

The line above uses the + operator to add the values stored in the locations `a` and `b` and the assignment operator (=) to store the result in the location `sum`. `a` and `b` are said to be the *operands* of +. The combination `a + b` is called an *expression*, specifically an *arithmetic expression* since + is an *arithmetic operator*. Similarly, = and its operands, `sum` and `a + b` together form the assignment expression `sum = a + b` (Note that the semicolon is not part of the expression). Other arithmetic operations that can be performed on integers (also common in many other languages) include:

- Subtraction, using the `-` operator
- Multiplication, using the `*` operator
- Division, using the `/` operator
- Remainder, using the `%` operator

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 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;
```

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;
```

In programming, when a certain pattern is used excessively, we tend to make a shortcut for it. Hence the compound assignment operators. They are

- +=
- -=
- *=
- /=
- %=
- <<=
- >>=
- |=
- &=
- ^=

They are shortcuts to the above pattern.

For example,

```
i += 1;
```

is equivalent to

```
i = i + 1;
```

The other compound assignment operators work in a similar way.

Thus the example given in the beginning of the section could be rewritten as

```
a += 1;
b *= 2;
c /= 4;
```

TODO
parent topic may need a re-writing.

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). 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 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.

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

Arrays store a constant-sized sequential set of blocks, each block containing a value of the elected type under a single name. Individual elements are accessed by their position in the array called its index, also known as subscript. It is easiest to think of an *array* as simply a list with each value as an item of the list. Arrays often help organize collections of data efficiently and intuitively. 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 use something like...

```
int numbers[30]; // creates an array of 30 int
```

or

```
char letters[4]; // create an array of 4 chars
```

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.

Access a value stored in an *array* is easy. For example with the above declaration,

```
int x;
x = vector[2];
```

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 after knowing about pointers.

To get an `array` of a different size, you must explicitly deal with memory using `realloc`, `malloc`, `memcpy`, etc.

Advantages of arrays include:

- Random access in O(1) (Big O notation)
- Ease of use/port: Integrated into most modern languages

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*.

Why no bounds checking on array indexes?

C++ does allow for (but doesn't force) bounds-checking implementations but in practice little or no checking is done, because it affects storage requirements (needing "fat pointers") and 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 "&"

This operator is used to get the address of a variable. It is also used when declaring reference types.

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`.

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
}
```

Dereferencing Operator ""*

This operator is used to get the variable pointed to by a pointer. It is also used when declaring pointer types.

Pointers

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. Essentially, they point to another variable. You can make a pointer point to a variable, and then when you pass it to a function, you let the function's pointer point to the same variable, so that it can access or modify it even if it couldn't otherwise. You can even have pointers to pointers, and pointers to pointers to pointers and so on and so forth.

Declaring Pointers

Pointers are declared by adding a `*` before the variable name in the declaration, as in the following example:

```
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 check the coding style conventions used and adder to a single use.

Watch out, though, because the * associates to the following declaration only:

```
int* i, j;  // WRONG! i is pointer to int, j is int.
int *i, *j; // RIGHT! 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).
```

Addressing 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.

```
int* x = &i;// must be used as rvalue
```

Now, do not confuse the "address of" operator with the declaration of a reference. Because use of operators is restricted to expression, the compiler knows that `&sometype` is the "address of" operator being used to denote the return of the address of `sometype` as a pointer.

Furthermore, if you have a function which has a pointer as an argument, you may use the "address of" operator on a variable to which you have not already set a pointer to point. By doing this, you do not necessarily have to declare a pointer just so that it is used as an argument in a function, the "address of" operator returns a pointer and thus can be used in that case too.

```
#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.

Dereferencing operators

Now that you have a pointer, you need some way to access the memory that it points to. This is the `*` operator. When it's 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 `*` operator with classes, you may notice a problem. It has lower precendence than `.`! See the example:

```
struct A { int num; };

A a;
int i;
A * p;

p = &a;
a.num = 2;

i = *p.a; // Error! "p" isn't a class, so you can't use "."
i = (*p).a;
```

It would be very time-consuming to have to write `(*p).a` 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).a`, you can write `p->a`, 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 very special pointer. 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 a macro defined in the standard library called `NULL`, which is always equal to a null pointer value (essentially, 0). It is good practice to always use `NULL` when referring to the null pointer, as it strongly improves readability.

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 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 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.

To define a pointer to a function, you have to define the function's return type and its parameters. These must be exact! 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), than 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,

as in the following example:

```
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(i)<<std::endl;    // Will output "g(4) is 8"
  std::cout<<"(*g)(5) is "<<g(i)<<std::endl; // Will output "g(5) is 10"
  return 0;
 }
```

TODO
Make short introduction to pointers as data members (so it can be cross linked from the function and class sections of the texts)

Pointer Indirection Operator "->"

This operator is used to access a member of a class pointer.

Pointer-to-Member Dereferencing Operator "."*

This operator is used to access the variable associated with a specific class instance, given an appropriate pointer.

Pointer-to-Member Indirection Operator "->"*

This 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.

sizeof()

The **sizeof** operator works at compile time to report on the number of bytes of storage occupied by a type (equivalently, by a variable of that type).

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 an object. Style guidelines vary on whether using the latitude to omit parentheses in the latter case is desirable.

sizeof has also found new life in recent years in template meta programming in C++, where the fact that it can turn types into numbers, albeit in a primitive manner, is often useful, given that the template metaprogramming environment of C++ typically does most of its calculations with types.

```
//Examples of sizeof use
std::size_t int_size( sizeof( int ) );// Might give 1, 2, 4, 8 or other values.

// or

int answer( 42 );
std::size_t answer_size( sizeof( answer ) );// Same value as sizeof( int )
std::size_t answer_size( sizeof answer);    // Equivalent syntax
```

Note that **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 number of bytes 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.

Dynamic Memory Allocation

Dynamic memory allocation is the allocation of memory storage for use in a w:computer program during the runtime 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 garbage collector implementation; this is notably different from automatic and static memory allocation, 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 while keeping both allocation and deallocation efficient. Also, the allocator's metadata can inflate the size of (individually) small allocations; chunking 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. 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 are can now be avoided but are still accessible for compatibility and low level control reasons.

TODO
add info on malloc

As covered before we use the assigning 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 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 sect 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 actions compiles and does nothing).

You must keep in mind that `new T` and `new T()` are not the 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 (RAII) idiom.

TODO
Move or split some of the information or add references, classes, destructor and constructors

have yet to be introduced and bellow we are using a vector for the example

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 runtime 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
```

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;
```

TODO
Add missing information on **Bitwise** and **Relational** operators.

Logical operators

The operators `&&` (**and**) and `||` (**or**) allow two or more conditions to be chained together. The `&&` operator checks whether all conditions are true and the `||` 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, determine how the checks are interpreted. Newer versions of the C++ standard, encourage the keywords **and** and **or** in place of && and ||. Both operators are said to *short circuit*. If a previous `&&` condition is false, later conditions are not checked. If a previous `||` condition is true later conditions are not checked.

The `!` (**not**) operator is used to return the inverse of one or more conditions.

- **Syntax**:

```
condition1 && condition2
condition1 || condition2
!condition
```

- **Semantic**:

if both condition1 and conditions2 are true the result is true else the result is false.

condition1	condition2	condition1 && condition2
true	true	true
true	false	false
false	true	false
false	false	false

if condition1 or condition2 is true the result is true else the result is false.

condition1	condition2	condition1 || condition2
true	true	true
true	false	true
false	true	true
false	false	false

if condition is true the result is false and if the condition is false the result is true.

condition	!condition
true	false
false	true

- **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) && !(x == 10)) // if (x grater 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 && 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 || x == 10 || 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 && x <= 20) || (x >= 30 and x <= 40)) // >= -> grater 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 `&&`. 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 Operator

Logical AND:

AND	**True**	**False**
True	True	False
False	False	False

The logical AND operator, **&&**, 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) && (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 **&&** is not the same as the address operator and the bitwise AND operator, both of which are represented with **&**

OR Operator

Logical OR:

OR	**True**	**False**
True	True	True
False	True	False

The logical OR operator is represented with ||. 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) || (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 Operator

The logical NOT operator, **!**, 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.

```
!x > 10
```

The logical expressions have a higher precedence than normal operators. Therefore, it compares whether "!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).

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.

- **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.".

TODO
Note the short-cut semantics of evaluation. Note the conditions on the types of the expressions, and the conversions that will be applied if they have different types. Note that code that discards the value of the conditional expression can be more clearly written using an **if** statement.

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**.

Type Conversion

Type conversion (often a result of **type casting**) refers to changing an entity of one data type 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 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 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 suggests that there isn't 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.

Automatic Type conversions

Whenever the compiler expects data of a particular type, but the data is given as a different type, it will try to automatically convert. (Note: this is not "casting"; casting is the use of specific notation in the source code to request a conversion or to specify a function from an overload set. There is no such thing as an "automatic cast".)

```
int a = 5.6;
float b = 7;
```

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 demotion. (Note: the term "demotion" is not normally used in the C++ community.

TODO
There are other implicit conversions to, such as derived-to-base conversions for pointers/references to objects of class types, and the implicit "decay" from an array to a corresponding pointer type.

Promotion

Promotion occurs whenever a variable or expression of a *smaller* type is converted to a *larger* type.

```
// promoting float to double...

float a = 4;    // 4 is a int constant, gets promoted to float
long b = 7;     // 7 is an int constant, gets promoted to long
double c = a;   // a is a float, gets promoted to double
```

There is generally no problem with automatic promotion. Programmers should just be aware that it happens.

Demotion

Demotion occurs whenever a variable or expression of a *larger* type gets converted to a *smaller* type. By default, a floating point number is considered as a double number in C++.

```
int a = 7.5;   // double gets down-converted to int;
int b = 7.0f;  // float gets down-converted to int;
char c = b;    // int gets down-converted to char;
```

Automatic demotion can result in the loss of information. In the first example the variable a will contain the value 7, since int variables cannot handle floating point values.

Most modern compilers will generate a warning if demotion occurs. 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.

Explicit type conversions (casting)

There are cases where no automatic type conversion can occur, where the compiler is unsure about what type to convert to, or other forms of typecasting are needed.

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.

Advanced type casts

const_cast

const_cast<T>(expression)
The const_cast<>() is used to add/remove **const**(ness) (or volatile-ness) of a variable.

static_cast

static_cast<T>(expression)
The static_cast<>() is used to cast between the integer types.
'e.g.' char->long, int->short etc.

Static cast is also used to cast pointers to related types, for example casting `void*` to the appropriate type.

dynamic_cast

Dynamic cast is used to convert pointers and references at run-time, generally for the purpose of casting a pointer or reference up or down an inheritance chain (inheritance hierarchy).

```
dynamic_cast<target type>(expression)
```

The target type must be a pointer or reference type, and the expression must evaluate to a pointer or reference. Dynamic cast works only when the type of object to which the expression refers is compatible with the target type and the base class has at least one virtual member function. If not, and the type of expression being cast is a pointer, NULL is returned, if a dynamic cast on a reference fails, a *bad_cast* exception is thrown. When it doesn't fail, dynamic cast returns a pointer or reference of the target type to the object to which expression referred.

reinterpret_cast

Reinterpret cast simply casts 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.

```
reinterpret_cast<target type>(expression)
```

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.

Older forms of type 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 style
l = long(i); //C++ style
```

The keyword casts are more controlled, and should generally be preferred. They're safer, and they're easier to search for in code.

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;
```

Summary of different casts

reinterpret_cast - mostly non-portable way to convert without changing the representation of a value

```
int a = 0xffe38024;
int * b = reinterpret_cast<int*>(a);
```

static_cast - pointer casts from base to derived class, or void* to *target type**

```
BaseClass* a = new DerivedClass();
static_cast<DerivedClass*>(a)->derivedClassMethod();
```

const_cast - changes a **const** qualifier

```
struct A { void func() {} };

void f(const A& a) {
   A& b = const_cast<A&>(a);
   b.func();
}
```

dynamic_cast - similar to **static_cast**, but has a runtime check which ensures that the object is really of the derived type you're casting to, and is also capable of navigating multiple inheritance hierarchies, including performing so-called "cross casts":

```
class A { ... };
```

```
class B : public A { ... };

void f(A* a) {
   B* b = dynamic_cast<B*>(a); // Won't compile
   B* b = static_cast<B*>(a);  // Will compile
}
class A { virtual void foo() {} };

class B : public A { ... };

void f(A* a) {
   B* b = dynamic_cast<B*>(a); // Will compile
   B* b = static_cast<B*>(a);  // Will compile
}
```

Control Flow Construct 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).

Logical Expressions as conditions

> Logical expressions can use logical operators in loops and conditional statements as part of the conditions to be met.

Conditionals

There is likely no meaningful program written in which a computer does not demonstrate basic decision-making skills. 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.

if (Fork branching)

The if-statement allows one possible path choice depending on the specified conditions.

Syntax

```
if (condition)
{
   statement;
}
```

Semantic

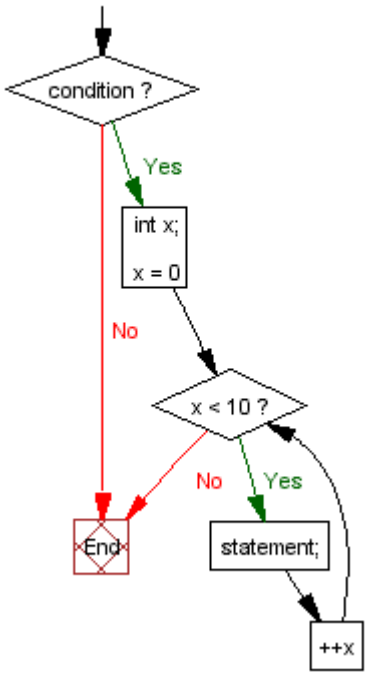
First, the condition is evaluated:

- 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.

Remark: The statements in an **if** statement can be any code that's valid; you can declare variables, nest statements, etc.

Example

```
if(condition)
{
   int x; // Valid code
   for(x = 0; x < 10; ++x) // Also valid.
      {
         statement;
      }
}
```



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.

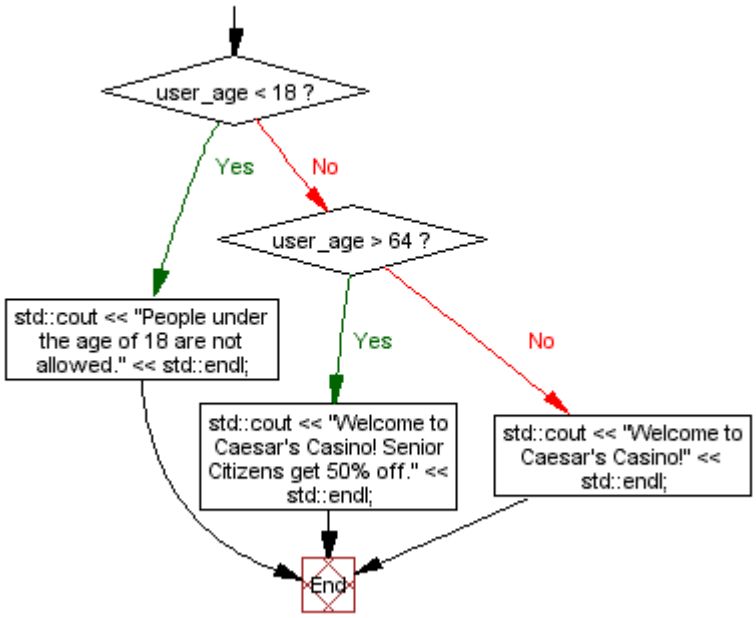
Sometimes the program needs to choose one of two possible paths depending on a condition. For this we can use the if-else statement.

```
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;
}
```



NOTE:

- **break** and **continue** don't have any relevance to an **if** or **else**. A **break** inside an **if**, will break past the loop or switch its in and **continue** will continue the loop it is in.
- 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)
            break;
}
```

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
             break;
}
```

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.

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 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 state.

When used in the first sense, recursion 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* (the "one" 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.

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.

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)

A special case of a Preconditional 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);
```

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;
}
```

or using the full syntax with a step of +1. **i** will grow from 0 by a factor of 1 in each loop until 9, as defined by the condition.

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.

Other Control Flow Constructs

goto

The **goto** statement is strongly discouraged as it makes it difficult to follow the program logic, this way inducing to errors. In some (mostly 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 exception handling is not a better option). 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.

Syntax

```
label:
   statement(s);

goto label;
```

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*.)

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;
```

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.

Functions

A **function** (which can also be referred to as subroutine, **procedure**, **subprogram** or even method) carries out tasks defined by a sequence of statements called a *statement block* 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*, and may pass results of a task on to the caller of the function, this is called the *return value*.

In C++ is important to note that a **function** that exists in the global scope 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 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's 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++.

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.

```
int main()
{
      // code
      return 0;
}
```

The above example defines 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).

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` is special in C++ in that 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 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.

Parameters

The syntax for declaring and invoking functions with multiple parameters is a common source of errors. First, remember that you have to declare the type of 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.

Another common source of confusion is that 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 hour = 11;
int minute = 59;
printTime( int hour, int minute ); // WRONG!
```

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. The correct syntax is `printTime( hour, minute )`.

Passing by Pointer

Arrays are similar to pointers, remember?

Now might be a good time to reread the section on arrays. If you don't 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 integers. 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 integer.

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
```

Look at that! We've passed in array elements just like normal integers! Because, of course, `my_array[2]` IS an integer.

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 do that too, of course.

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
```

Look at that, we've passed a whole array to a function! Now here's some important points to realize:

- Once you pass an array to a function, 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` doesn't 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`.

TODO
Passing a single element (by value vs. by reference), passing the whole array (always by reference), passing as **const**

Passing 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:
While sometimes useful, using this style of references can sometimes lead to counter-intuitive code. It is not clear to the caller of foo() above that bar will be modified without consulting an API reference. Choosing expressive names for functions helps to make the meaning clearer, as always.

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 wtih indirection may make code using a reference slower than code that copies small objects.

Passing 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.

Read the following code:

```
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
```

In function "add", we can see that the value of "num1" is changed in line 3. However, we can also see that the value of "a" keeps after passed to this function. This shows a property of pass-by-value, the arguments are copies of the passed variable and only in the scope of the corresponding function.

NOTE:
Technically speaking, when a variable is passed by value, the copy constructor of the corresponding type (in the above case, "std::string") will be called, and copy the value/member variables of that variable to a new variable (in the above case, the value of "text" is copied to "s" in function "bar"). So, we have to afford the cost of copying, but we usually only consider this when passing complex variables. This is also the reason why the change of value of the arguments will not affect the passed variable when passed by value.

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 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 doesn't 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.

Returning Values

When declaring a function, you must declare it in terms of the type that it will return, 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;
}
```

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.

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. 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 return values," and we'll do it in a couple of chapters. 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 can not rely on the compiler for two reasons: First a compiler has 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 reason 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!

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 you are currently working in. However, there may be valid reasons for making a different choice in any particular situation.

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
      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 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.

One advantage here is that you follow the structured programming principle of a function having a single entry and a single exit.

The Microsoft Foundation Class Library (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 (my_function1())
{
      //error handler for function 1 errors
      return FUNCTION_1_FAILED;
}
// block of instruction 1
if (my_function2())
{
      //error handler for function 2 errors
      return FUNCTION_2_FAILED;
}
// block of instruction 2
if (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 (libc) is an example of a standard library that uses this paradigm.

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.

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.

Recursion

In programming languages, recursion was first implemented in Lisp 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.

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 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 nth Fibonacci number using recursion.

Inline

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, macros can lead to problems since they aren't evaluated by the preprocessor. 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, this will be covered on the **Object-Oriented Programming Chapter** .

TODO
Complete and give examples

Pointers to functions

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.

Example

```
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>)*.

Example

```
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.
```

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;
```

Overloading

In C++ you can define and use different functions with the same name. 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.

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 don't despair if you do not understand this at first go. It's really quite natural, once you have the ideas, but written down it seems extremely complicated.

TODO

- This section does not cover the selection of constructors because, well, that's even worse. Namespaces are also not considered below.
- Feel free to add the missing information, possibly as another chapter.

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 parameterss 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 don't 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
    1. **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
    1. Base to derived (pointers) or derived to base (for pointers-to-members), with most-derived preferred
    2. Conversion to `void*`
    3. 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 visible**

If the selected best function is not visible (e.g, private), the call fails.

Standard C Library

The **C standard library** is a standardized collection of header files and library routines used to implement common operations, such as input/output and string handling. It became part of the C++ Standard Library as the **Standard C Library** in its ANSI C 89 form with some small modifications to make it work better with the C++ language.

For a more in depth look into the C programming language check the C Programming Wikibook but be aware of the incompatibilities we have already covered on the Comparing C++ with C Section of this book.

Standard C Library Functions (All)

Functions	Descriptions
abort	stops the program
abs	absolute value
acos	arc cosine
asctime	a textual version of the time
asin	arc sine
assert	stops the program if an expression isn't true
atan	arc tangent
atan2	arc tangent, using signs to determine quadrants
atexit	sets a function to be called when the program exits
atof	converts a string to a double

atoi	converts a string to an integer
atol	converts a string to a long
bsearch	perform a binary search
calloc	allocates and clears a two-dimensional chunk of memory
ceil	the smallest integer not less than a certain value
clearerr	clears errors
clock	returns the amount of time that the program has been running
cos	cosine
cosh	hyperbolic cosine
ctime	returns a specifically formatted version of the time
difftime	the difference between two times
div	returns the quotient and remainder of a division
exit	stop the program
exp	returns "e" raised to a given power
fabs	absolute value for floating-point numbers
fclose	close a file
feof	true if at the end-of-file
ferror	checks for a file error
fflush	writes the contents of the output buffer
fgetc	get a character from a stream
fgetpos	get the file position indicator
fgets	get a string of characters from a stream

floor	returns the largest integer not greater than a given value
fmod	returns the remainder of a division
fopen	open a file
fprintf	print formatted output to a file
fputc	write a character to a file
fputs	write a string to a file
fread	read from a file
free	returns previously allocated memory to the operating system
freopen	open an existing stream with a different name
frexp	decomposes a number into scientific notation
fscanf	read formatted input from a file
fseek	move to a specific location in a file
fsetpos	move to a specific location in a file
ftell	returns the current file position indicator
fwrite	write to a file
getc	read a character from a file
getchar	read a character from STDIN
getenv	get enviornment information about a variable
gets	read a string from STDIN
gmtime	returns a pointer to the current Greenwich Mean Time
isalnum	true if a character is alphanumeric
isalpha	true if a character is alphabetic
iscntrl	true if a character is a control character

isdigit	true if a character is a digit
isgraph	true if a character is a graphical character
islower	true if a character is lowercase
isprint	true if a character is a printing character
ispunct	true if a character is punctuation
isspace	true if a character is a space character
isupper	true if a character is an uppercase character
isxdigit	true if a character is a hexidecimal character
labs	absolute value for long integers
ldexp	computes a number in scientific notation
ldiv	returns the quotient and remainder of a division, in long integer form
localtime	returns a pointer to the current time
log	natural logarithm
log10	natural logarithm, in base 10
longjmp	start execution at a certain point in the program
malloc	allocates memory
memchr	searches an array for the first occurance of a character
memcmp	compares two buffers
memcpy	copies one buffer to another
memmove	moves one buffer to another
memset	fills a buffer with a character
mktime	returns the calendar version of a given time
modf	decomposes a number into integer and fractional parts

perror	displays a string version of the current error to STDERR
pow	returns a given number raised to another number
printf	write formatted output to STDOUT
putc	write a character to a stream
putchar	write a character to STDOUT
puts	write a string to STDOUT
qsort	perform a quicksort
raise	send a signal to the program
rand	returns a pseudorandom number
realloc	changes the size of previously allocated memory
remove	erase a file
rename	rename a file
rewind	move the file position indicator to the beginning of a file
scanf	read formatted input from STDIN
setbuf	set the buffer for a specific stream
setjmp	set execution to start at a certain point
setlocale	sets the current locale
setvbuf	set the buffer and size for a specific stream
signal	register a function as a signal handler
sin	sine
sinh	hyperbolic sine
sprintf	write formatted output to a buffer
sqrt	square root

srand	initialize the random number generator
sscanf	read formatted input from a buffer
strcat	concatenates two strings
strchr	finds the first occurance of a character in a string
strcmp	compares two strings
strcoll	compares two strings in accordance to the current locale
strcpy	copies one string to another
strcspn	searches one string for any characters in another
strerror	returns a text version of a given error code
strftime	returns individual elements of the date and time
strlen	returns the length of a given string
strncat	concatenates a certain amount of characters of two strings
strncmp	compares a certain amount of characters of two strings
strncpy	copies a certain amount of characters from one string to another
strpbrk	finds the first location of any character in one string, in another string
strrchr	finds the last occurance of a character in a string
strspn	returns the length of a substring of characters of a string
strstr	finds the first occurrence of a substring of characters
strtod	converts a string to a double
strtok	finds the next token in a string
strtol	converts a string to a long
strtoul	converts a string to an unsigned long
strxfrm	converts a substring so that it can be used by string comparison

	functions
system	perform a system call
tan	tangent
tanh	hyperbolic tangent
time	returns the current calendar time of the system
tmpfile	return a pointer to a temporary file
tmpnam	return a unique filename
tolower	converts a character to lowercase
toupper	converts a character to uppercase
ungetc	puts a character back into a stream
va_arg	use variable length parameter lists
vprin, vfprintf, and vsprintf	write formatted output with variable argument lists

These routines included on the Standard C Library can be sub divided into:

- Standard C I/O
- Standard C String & Character
- Standard C Math
- Standard C Time & Date
- Standard C Memory
- Other standard C functions

Standard C I/O

The Standard C Library includes routines that are somewhat outdated, but due to the history of the C++ language 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. When an error occurs, you can use perror() to figure out which error actually occurred.

Related topics

feof - ferror - perror

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 - fopen - freopen - setbuf

feof

Syntax

```
#include <cstdio>
int feof( FILE
*stream );
```

The function abort() terminates the current program. Depending on the implementation, the return value can indicate failure.

Related topics

clearerr - ferror - getc - perror - putc

ferror

Syntax

```
#include <cstdio>
int ferror( FILE
*stream );
```

The ferror() function looks for errors with stream, returning zero if no errors have occured, and non-zero if there is an error. In case of an error, use perror() to determine which error has occured.

Related topics

clearerr - feof - perror

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, then fflush() causes the input buffer to be cleared. fflush() is useful when debugging, if a program segfaults before it has a chance to write output to the screen. Calling fflush(**stdout**) directly after debugging output will ensure that your output is displayed at the correct time.

```
printf( "Before first call\n" );
fflush( stdout );
shady_function();
printf( "Before second call\n" );
fflush( stdout );
dangerous_dereference();
```

Related topics

fclose - fopen - fread - fwrite - getc - putc

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 - fputc - fread - fwrite - getc - getchar - gets - putc

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 - fsetpos - ftell

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

fputs - fscanf - gets - scanf

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)

Mode	Meaning
"r"	Open a text file for reading
"w"	Create a text file for writing
"a"	Append to a text file
"rb"	Open a binary file for reading
"wb"	Create a binary file for writing
"ab"	Append to a binary file
"r+"	Open a text file for read/write
"w+"	Create a text file for read/write
"a+"	Open a text file for read/write
"rb+"	Open a binary file for read/write
"wb+"	Create a binary file for read/write
"ab+"	Open a binary file for read/write

An example:

```
int ch;
FILE *input = fopen( "stuff", "r" );
ch = getc( input );
```

Related topics

fclose - fflush - fgetc - fputc - fread - freopen - fseek - fwrite - getc - getchar - setbuf

fprintf

Syntax

```
#include <cstdio>
int fprintf( FILE *stream, const char *format,
... );
```

The fprintf() function sends information (the arguments) according to the specified format to the file indicated by stream. fprintf() works just like printf() 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 - fputs - fscanf - printf - sprintf

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

fgetc - fopen - fprintf - fread - fwrite - getc - getchar - putc

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 - fprintf - fscanf - gets - getc - puts

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() or ferror() to figure out if an error occurs.

Related topics

fflush - fgetc - fopen - fputc - fscanf - fwrite - getc

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 access given by mode. The return value of freopen() is the new stream, or **NULL** if there is an error.

Related topics

fclose - fopen

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, or **EOF** if no assignments could be made.

Related topics

fgets - fprintf - fputs - fread - fwrite - scanf - sscanf

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 - fopen - fsetpos - ftell - rewind

fsetpos

Syntax

```
#include <cstdio>
int fsetpos( FILE *stream, const fpos_t
*position );
```

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 - fseek - ftell

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 - fseek - fsetpos

fwrite

Syntax

```
#include <cstdio>
int fwrite( const void *buffer, size_t size, size_t count, FILE
*stream );
```

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 - fgetc - fopen - fputc - fread - fscanf - getc

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(). For example:

```
int ch;
FILE *input = fopen( "stuff", "r" );

ch = getc( input );
while( ch != EOF ) {
   printf( "%c", ch );
   ch = getc( input );
}
```

Related topics

feof - fflush - fgetc - fopen - fputc - fgetc - fread - fwrite - putc - ungetc

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 - fopen - fputc - putc

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().

Related topics

fgetc - fgets - fputs - puts

perror

Syntax

```
#include <cstdio>
void perror( const char
*str );
```

The perror() function prints str and an implementation-defined error message corresponding to the global variable errno. For example:

```
char* input_filename = "not_found.txt";
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
```

Related topics

clearerr - feof - ferror

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";
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

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 - puts - scanf - sprintf

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 );
}
fclose( input );
fclose( output );
```

Generates a copy of the file tmp.c called tmpCopy.c.

Related topics

feof - fflush - fgetc - fputc - getc - getchar - putchar - puts

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

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 - gets - printf - putc

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

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

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

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(). 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.

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 - fscanf - printf - sscanf

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. If a non-standard buffer size is used, it should be BUFSIZ characters long.

Related topics

fclose - fopen - setvbuf

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:

- _IOFBF, which indicates full buffering
- _IOLBF, which means line buffering
- _IONBF, which means no buffering

Related topics

setbuf

sprintf

Syntax

```
#include <cstdio>
int sprintf( char *buffer, const char *format,
... );
```

The sprintf() function is just like printf(), 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() -- where atoi() 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 - printf
(Standard C String and Character) atof - atoi - atol

sscanf

Syntax

```
#include <cstdio>
int sscanf( const char *buffer, const char *format, ...
);
```

The function sscanf() is just like scanf(), except that the input is read from *buffer*.

Related topics

fscanf - scanf

tmpfile

Syntax

```
#include <cstdio>
FILE *tmpfile(
void );
```

The function tmpfile() opens a temporary file with an unique filename and returns a pointer to that file. If there is an error, null is returned.

Related topics

tmpnam

tmpnam

Syntax

```
#include <cstdio>
char *tmpnam( char
*name );
```

The tmpnam() function creates an unique filename and stores it in name. tmpnam() can be called up to **TMP_MAX** times.

Related topics

tmpfile

ungetc

Syntax

```
#include <cstdio>
int ungetc( int ch, FILE
*stream );
```

The function ungetc() puts the character *ch* back in *stream*.

Related topics

getc
(C++ I/O) putback

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(), fprintf(), and sprintf(). 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(). 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 );
}
```

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];
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 - atol - strtod
(Standard C I/O) sprintf

atoi

Syntax

```
#include <cstdlib>
int atoi( const char
*str );
```

The atoi() function converts str into an integer, and returns that integer. *str* should start with 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 - atol
(Standard C I/O) sprintf

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.

Related topics

atof - atoi - strtod
(Standard C I/O) sprintf

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 - iscntrl - isdigit - isgraph - isprint - ispunct - isspace - isxdigit

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 - iscntrl - isdigit - isgraph - isprint - ispunct - isspace - isxdigit

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

isalnum - isalpha - isdigit - isgraph - isprint - ispunct - isspace - isxdigit

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 - isalpha - iscntrl - isgraph - isprint - ispunct - isspace - isxdigit

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 - isalpha - iscntrl - isdigit - isprint - ispunct - isspace - isxdigit

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

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 - isalpha - iscntrl - isdigit - isgraph - ispunct - isspace

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

isalnum - isalpha - iscntrl - isdigit - isgraph - isspace - isxdigit

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 - isalpha - iscntrl - isdigit - isgraph - isprint - ispunct - isxdigit

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 - tolower

isspace

Syntax

```
#include <cctype>
int isxdigit( int
ch );
```

The function isxdigit() returns non-zero if its argument is a hexidecimal digit (i.e. A-F, a-f, or 0-9). Otherwise, zero is returned.

Related topics

isalnum - isalpha - iscntrl - isdigit - isgraph - ispunct - isspace

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 - memcpy - strstr

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*
greater than 0	*buffer1* is greater than *buffer2*

Related topics

memchr - memcpy - memset - strcmp

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 - memcmp - memmove - memset - strcpy - strlen - strncpy

memmove

Syntax

```
#include <cstring>
void *memmove( void *to, const void *from, size_t count
);
```

The memmove() function is identical to memcpy(), except that it works even if *to* and *from* overlap.

Related topics

memcpy - memset

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.

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 - memcpy - memmove

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().

Related topics

strchr - strcmp - strcpy - strncat

strchr

Syntax

```
#include <cstring>
char *strchr( const char *str, int
ch );
```

The function strchr() returns a pointer to the first occurence of *ch* in *str*, or **NULL** if *ch* is not found.

Related topics

strcat - strcmp - strcpy - strlen - strncat - strncmp - strncpy - strpbrk - strspn - strstr - strtok

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 - strcat - strchr - strcoll - strcpy - strlen - strncmp - strxfrm

strcoll

Syntax

```
#include <cstring>
int strcoll( const char *str1, const char
*str2 );
```

The strcoll() function compares str1 and str2, much like strcmp(). However, strcoll() performs the comparison using the locale specified by the (Standard C Date & Time) setlocale() function.

Related topics

strcmp - strxfrm
(Standard C Date & Time) setlocale

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().

Related topics

memcpy - strcat - strchr - strcmp - strncmp - strncpy

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 - strrchr - strstr - strtok

strerror

Syntax

```
#include <cstring>
char *strerror( int
num );
```

The function strerror() returns an implementation defined string corresponding to *num*.

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 - strchr - strcmp - strncmp

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.

Related topics

strcat - strchr - strncmp - strncpy

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 - strcmp - strcpy - strlen - strncat - strncpy

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*. If from has less than *count* characters, the remainder is padded with '\0' characters. The return value is the resulting string.

Related topics

memcpy - strchr - strcpy - strncat - strncmp

C++ Programming/Code/Standard C Library/Functions/strpbrk

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

strcspn - strpbrk - strspn - strstr - strtok

strspn

Syntax

```
#include <cstring>
size_t strspn( const char *str1, const char *str2
);
```

The strspn() function returns the index of the first character in *str1* that doesn't match any character in *str2*.

Related topics

strchr - strpbrk - strrchr - strstr - strtok

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

memchr - strchr - strcspn - strpbrk - strrchr - strspn - strtok

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

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 - strcspn - strpbrk - strrchr - strspn - strstr

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 - strtoul

strtoul

Syntax

```
#include <cstdlib>
unsigned long strtoul( const char *start, char **end, int base
);
```

The function strtoul() behaves exactly like strtol(), except that it returns an unsigned long rather than a mere long.

Related topics

strtol

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() is performed on *str1* and the old *str2*, you will get the same result as with a strcmp().

Related topics

strcmp - strcoll

tolower

Syntax

```
#include <cctype>
int tolower( int
ch );
```

The function tolower() returns the lowercase version of the character *ch*.

Related topics

isupper - toupper

toupper

Syntax

```
#include <cctype>
int toupper( int
ch );
```

The toupper() function returns the uppercase version of the character *ch*.

Related topics

tolower

Debugging

Debugging is a fundamental skill of any programmer. Debugging can be quite stressful if you are under pressure (time constrains) but it can also be quite fun, 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. Other alternatives are running automated tools to test or verify the code or analise the code as it runs, this is the task were a debugger can came to your aid.

What is debugging?

Programming is a complex process, and since it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called bugs 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.

There are a few different kinds of errors that can occur in a program, and it is useful to distinguish between them in order to track them down more quickly.

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 pointed out, "When you have eliminated the impossible,

whatever remains, however improbable, must be the truth." (from A. Conan Doyle's 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 is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds 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 The Linux Users' Guide Beta Version 1, Page 10).

Type of errors

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 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 complier/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 second type of error is a run-time error, so-called because the error does not appear until you run the program.

For the simple sorts of programs we will be writing for the next few weeks, run-time errors are rare, so it might be a little while before you encounter one.

Logic errors and semantics

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.

Common errors

- forgetting to check for null before accessing a member on a pointer. This will cause access violations (segmentation faults) and cause your program to halt unexpectedly. Example:

```
// unsafe
p->doStuff();

// much better!
if (p)
{
    p->doStuff();
}
```

Typographical errors (typos)

- Assignment of a number in an if statement when a comparison was meant.

```
if ( x = 143 ) // should be: if ( x == 143)
```

- Forgetting the `break` statement in a `switch` when fall-through was not meant
- Forgetting the `;` at the end of a line. All time classic !
- Forgetting the brackets in a multi lined loop.

```
if (x==3)
cout << x;
flag++;
```

Debugger

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.

It's 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.

What is a 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.

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 chosing.
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.

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 (interactive 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's 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 don't 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 don't 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) in the margin to indicate there is a breakpoint at that point in the code.

Other runtime analyzers

Chapter Summary

1. Compiler - Introduction, list of recognized *keywords* and its directives (**inline**, **static**, etc...).
    1. Preprocessor - includes *the standard headers*.
    2. Linker
2. Internal storage of data types - *bits* and *bytes*, *data* versus *variables*, *two's complement*, *endian* and *floating point*.
3. Variables - Introduction (*declaration*, *assignment*, *scope* and *visibility*), *primitive types*, *enumeration* with source examples.
4. Scope - with source examples.
    1. Namespace
5. Operators - *precedence order* and *composition*, , *assignment*, **sizeof**, **new**, **delete**, **[]** (*arrays*), ***** (*pointers*), **&** (*references*) and brief mention of *operator overloading*.
    1. Logical - the **&&** (and), || (or), and **!** (not) *Operators*.
    2. Conditional - the **?:** *Operator*.
6. Type casting - *Automatic*, *explicit* and *advanced type casts*.
7. Flow of control - Conditionals (**if**, **if-else**, **switch**), loop iterations (**while**, **do-while**, **for**) and **goto**.
8. Functions - Introduction (including **main()**), *argument passing*, *returning values*, *recursive functions*, *pointers to functions* and *function overloading*.
9. Standard C Library
    1. I/O
    2. String & Character
10. Debugging - Detect, correct and prevent bugs.

Object Oriented Programming

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.

A struct is *defined* by:

```
struct myStructType /*: inheritances */ {
public:
  // public members
protected:
  // protected members
private:
  // private members
} myStructName;
```

It is not common to have structs using inheritances but they are supported just like in classes, the reason this in uncommon is probably because this is specific to C++, and not supported in C. 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 is public.

An object of type *myStruct* (case-sensitive) is *declared* using:

```
myStruct obj1;
```

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 (ie: 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) 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; };
```

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:

```
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 = blank.x * blank.x + blank.y * blank.y;
```

The first line outputs 3, 4; the second line calculates the value 25.

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 squiggly braces 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. I'm sorry. (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.

Getting user input

The programs we have written so far are pretty predictable; they do the same thing every time they run. Most of the time, though, we want programs that take input from the user and respond accordingly.

There are many ways to get input, including keyboard input, mouse movements and button clicks, as well as more exotic mechanisms like voice control and retinal scanning. In this text we will consider only keyboard input.

In the header file `iostream`, C++ defines an object named cin that handles input in much the same way that cout handles output. To get an integer value from the user:

```
int x;
cin >> x;
```

The >> operator causes the program to stop executing and wait for the user to type something. If the user types a valid integer, the program converts it into an integer value and stores it in x.

If the user types something other than an integer, the compiler doesn't report an error, or anything sensible like that. Instead, it leaves the old (probably meaningless value) in x and continues.

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;
```

Unfortunately, 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.

Because of these problems (inability to handle errors and funny behavior), I avoid using the >> operator altogether, unless I am reading data from a source that is known to be error-free.

Instead, I use a 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.)

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.

Nesting structures

Structures can also be nested so that a valid element of a structure can also be another structure.

this

The this keyword is a 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. The this pointer is not available in static member functions. This will be restated again on when introducing unions a more indepth analysis is provided in the Section about classes.

Unions

Unions are similar to `structs`, but they differ in one aspect: the fields of a union share the same position in memory. 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.

Writing to Different Bytes

Unions are mostly 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.

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.

this

The this keyword is a implicitly created pointer that is only accessible within nonstatic 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 indepth analysis is provided in the Section about classes.

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*" 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.

In C++ the standard there isn't a definition for method, when discussing with users of other languages, the use of method as representing 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.

Declaration

A class is *defined* by:

```
class myClass {
  /* public, protected and private
  variables, constants, and functions */
};
```

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.

Inheritance (Derivation)

As we have seen early on the Programming Paradigms Section, Inheritance is a property that describes a relationship between two (or more) types, or classes, of objects in OOP and so C++ classes share this property.

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 form another or even several (Multiple inheritance), 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 anther 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
   }

};

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;
```

Now all is fine and supe'cool at the moment, except for one thing that you can easily notice: there are certain chunks of code that are very similar (if not exactly the same) in the above two classes. So, in essence, you have to type in the same code at two different places! 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 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 "publically" 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( 2*(3.14), 2));
   }

   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:

Access specifiers in the base class

	private	public	protected
public inheritance	The member is inaccessible.	The member is public.	The member is protected.
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.

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.

Lets 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 is the construction in which objects inherits from more than one object type or class. This contrasts with single inheritance, where a object can 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 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" Most modern OOP languages do not allow multiple inheritance.

Access Labels

C++ supports three labels (**Public**, **Protected** and **Private**) 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. A 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 - *C2248*), 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 fill; // dl of liquid

   public:
      Bottle()  // Default Constructor
      : fill(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 (fill>0)
          {
             --fill;
             return true;
```

```
          }
          else
          {
             return false;
          }

       }

       int level() const  // return level of liquid dl
       {
             return fill;
       }
 };  // 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

This is used for members that cannot be used in either subclasses or other places. This is 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.

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_value = f.m_value; // perfectly legal
```

```
    }

 private:
    int m_value;
}
```

protected

> The protected label has a special meaning related to inheritance. In the section on inheritance we will see more about it. Other that, it is just like the private label: only member functions can access members declared in a protected scope.

Data Members

TODO
Add missing info

this pointer

this is a C++ keyword. The *this* pointer acts like any other pointer. Read the section concerning pointers and references to understand more about what a pointer does. The *this* pointer is only accessible within nonstatic member functions of a **class** (or a **union** or **struct**). The *this* pointer is not available in static member functions. It is not necessary for the programmer to write code for the *this* pointer as the compiler does this implicitly. When you debug code in a GUI compiler with some way of viewing the current variables and when the program steps into nonstatic class functions, you can see the *this* pointer in some variable list.

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 x;
public:
      Foo(){  x=5;  };
      int getData(this)  // this is provided by the compiler at compile time
      {
            return this->x;
      };
};

int main()
{
      Foo Example;
      int temp;

      temp = Example.getData(&Example);  // compiler adds the &Example reference at compile time
      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 x;
public:
      Foo(){  x=5;  };
```

```
      int getData()
      {
            return x;
      };
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->x = RHS.x;        // 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*, the pointer is important in implementing any class.

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*" 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 "**::**". 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
// teach/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"

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;
}
```

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.

Take for example:

```
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;
 };
```

static

Member functions or variables declared static are shared between all instances of an object type. This means that only one copy of the member function or variable exists for any object type. 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 `C` 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.
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 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);
}
```

Inline

Sharing most of the concepts we have seen before on the introduction to inline functions, when dealing with member function those concepts are extended, with a few additional considerations.

If the member functions definition is included inside the definition of the class. that function is by default made implicitly inline. Compiler options may also 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.

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 operations.
>
> **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 an modifier due to the naming (`SetSize( int a_Size )`) of that type of member functions.

Lazy initialization

It is always needed to maintain the balance between the performance of the system and the resource consumption. Let us see how to reduce the resource consumption until it is required.

Lazy instantiation is one of the memory conservation mechanism, by which the object initialization is deferred until it is required.

Look at the following example:

```
class Wheel {
            int speed;
      public:
            int getSpeed(){
                  return speed;
            }
            void setSpeed(int speed){
                  this->speed = speed;
            }
 };

 //this must be corrected
 class Car{
        private Wheel wheel = new Wheel();
        public int getCarSpeed(){
            return wheel.getSpeed();
        }
        public String getName(){
```

```
            return "My Car is a Super fast car";
        }
        public static void main(String a[]){
           Car myCar = new Car();
           System.out.println(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. Here the instance wheel doesn't serve any purpose, loading of which 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:

```
//must be corrected
 class Car {
        private Wheel wheel;
        public int getCarSpeed(){
            if(wheel == null){
                wheel = new Wheel();
            }
            return wheel.getSpeed();
        }
        public String getName(){
            return "My Car is a Super fast car";
        }
        public static void main(String a[]){
           Car myCar = new Car();
           System.out.println(myCar.getName());
        }
 }
```

Now the Wheel will be instantiated only, when the member function getCarSpeed() is called. This is known as Lazy initialization.

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, 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 as any normal member function but it will share the name of the class. 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 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.

Example when declaring an object of type *class Foo* (Foo defined in 'Understanding Member Functions' above):

```
Foo myTest;                 // essentially what happens is:  Foo myTest();
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, if an exception is thrown before a constructor is completed, the associated destructor will not be called for that object, see more about it in C++ Programming/Exception Handling.

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 is to always add the members to the initializer list in the same order they're declared.

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 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.

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.

As seen previously in the Programming Paradigms Section, Polymorphism 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 a some considerations when we do this, so we must introduce now 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` functions are an essential part of designing a class hierarchy and sub-classing classes from a toolkit. The concept is relatively simple, but often misunderstood. Specifically it determines the behavior of overridden methods in certain contexts.

By placing the keyword `virtual` before a method declaration we are indicating that when we 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 still good style to do so.

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.

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.

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. 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 (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 destructor 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)
```

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.

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:
                     bool isAlive;
                     animalType Type;
                     int numberOfChildren;
   }

   class mammal : public animal{
           public:
                     int numberOfTeets;
   }

   class cat: public mammal{
           public:
                    bool likesFish; //probably true
   }

   int main {
         animal* a1= new animal;
         animal* a2= new mammal;
         animal* a3= new cat;
         mammal* m= new cat;

         a2->isAlive= True; //Correct
         a2->Type= Herbivore; //Correct
         m->numberOfTeets=2; //Correct

         a2->numberOfTits=6; //Incorrect
         a3->likesFish=True; //Incorrect

         cat* c= (cat*)a3; //Downcast, correct (but very poor practice, see later)
         c->likesFish=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* c = new cat;
motorbike* m;
m = dynamic_cast<motorbike*>(c);
```

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.

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 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. 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. One of the Boost libraries, contains the `boost:noncopyable` utility class which is a practical example. Below shows an example of using this class:

```
class C : boost::noncopyable
{
   ...
};
```

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 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.

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 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.

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, an 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;
    virtual void Save(ostream&) const = 0;
 };

 class Triangle : public DrawableObject
```

```
{
public:
   void Draw(GraphicalDrawingBoard&) const;
   void Save(ostream&) const;
};

class Rectangle : public DrawableObject
{
public:
   void Draw(GraphicalDrawingBoard&) const;
   void Save(ostream&) const;
};

class Circle : public DrawableObject
{
public:
   void Draw(GraphicalDrawingBoard&) const;
   void Save(ostream&) const;
};

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::const_iterator iter = drawableList.begin(),
     endIter = drawableList.end();
     iter != endIter;
     ++iter)
{
   (*iter)->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 and 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 "isa" vs "hasa", as in a Square is a Rectangle, but a Square has a set of sides.

What is a "nice" class?

A "nice" class takes into consideration the use of the following functions:

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 (STL), that will 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.

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;
```

```
}
```

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);
}
```

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.

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.

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.

Operator overloading

Operator overloading (less commonly known as ad-hoc polymorphism) is a specific case of polymorphism (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. It can easily be emulated using function calls:

```
:a + b × c
```

In a language that supports operator overloading is effectively a more concise way of writing:

```
operator_add (a, operator_multiply (b,c))
```

(Assuming the × operator has higher precedence than +.)

Operator overloading provides more than an aesthetic benefit when the language allows operators to be invoked implicitly in some circumstances.

Operator overloading has been criticized because it allows programmers to give operators completely different functionality depending on the types of their operands, 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@(parameter_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.

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.compare(name)==0);
      }
};
```

NOTE:
The `inline` keywords in the example above are technically redundant, as functions defined within a class definition like this are implicitly inline.

Operators as member functions

Operators may be overloaded as member or non-member functions. Aside from the operators which must be members, 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 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) {...}

      // binary operator as non-member function
      Vector2D operator+(const Vector2D& left, const Vector2D& right) {...}

      // unary operator as member function
      Vector2D Vector2D::operator-() {...}

      // unary operator as non-member function
      Vector2D operator-(const Vector2D& vec) {...}
```

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 operator+(const Vector2D& left, const Vector2D& right)
{
      Vector2D result;
      result.set_x(left.x() + right.x());
      result.set_y(left.y() + right.y());
      return result;
}
```

It is good style to only overload these operators to perform their customary arithmetic operation.

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>>( 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 WithRawPointer {
      T *m_ptr;
      public:
      WithRawPointer(T *ptr) : m_ptr(ptr) {}
      WithRawPointer& operator=(WithRawPointer const &rhs) {
            delete m_ptr; // free resource;
            m_ptr = 0;
            m_ptr = rhs.m_ptr;
            return *this;
      };
};

WithRawPointer x(new T);
x = x; // x.m_ptr == 0.

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.

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)
{
      return (*this + right);

}
```

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 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

- **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
```

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 know the reason why we cannot overload above operators refer to "Why can't I overload dot, ::, sizeof, etc.?" see www.research.att.com (http://www.research.att.com/~bs/bs_faq2.html#overload-dot).

Chapter Summary

1. Structures
2. Unions
3. Classes (Inheritance , Member Functions, Polymorphism and this pointer)
    1. Abstract Classes including Pure abstract classes (abstract types)
    2. Nice Class
4. Operator overloading

Advanced Features

I/O

Also commonly referenced as the C++ I/O of the C++ standard since the standard also includes the C Standard library and its I/O implementation, as seen before in the Standard C I/O Section.

The <iostream> library automatically defines a few standard objects:

- cout, an object of the ostream class, which displays data to the standard output device.
- cerr, another object of the ostream class that writes unbuffered output to the standard error device.
- clog, like cerr, but uses buffered output.
- cin, an object of the istream class that reads data from the standard input device.

The <fstream> library allows programmers to do file input and output with the ifstream and ofstream classes.

TODO
Remember to complete with the String Stream class and include the various io stream format flags.

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:

```
using namespace std;
string std_string;
```

or

```
std::string std_string;
```

Text I/O

Perhaps the most basic use of the string class is for reading text from the user and writing it to the screen, such as in the following code fragment:

```
std::string name;
std::cout << "Please enter your first name: ";
std::cin >> name;
std::cout << "Welcome " << name << "!";
```

Although a string may hold a sequence containing any character--including spaces--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);
```

More advanced string manipulation

TODO
Detail the commonly used std::string member functions(partially done)

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, ie. 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

```
int string.size(void);
int string.length(void);
```

So for example one might do:

```
int strSize =  str.size(void);
int strSize2 = str2.length(void);
```

The functions `size()` and `length()` both return the size of the parent string. 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 (ie 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[](int pos)
```

`Chars` in `strings` 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, similiar to the C strcmp() function. These return a true/false value.

```
if(str == "Hello World!")
{
  std::cout << "Strings are equal!";
}
```

Searching strings

```
int string::find(string findstr, int pos);
```

You can use the `find()` member function to find the first occurrence of a string inside another. `find()` will look for `findstr` inside `this` starting from position `pos` and return the position of the first occurrence of findstr. For example:

```
std::string str = "Hello World!";
std::string find = "o";
std::cout << str.find(find, 0);
```

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(int i = 0, tfind; (tfind = wikistr.find("wiki", i)) != string::npos; i = tfind + 1)
{
  std::cout << "Found occurrence of 'wiki' at position " << tfind << std::endl;
}
```

```
int string::rfind(string findstr, int pos);
```

The function `rfind()` works similarly, except it returns the *last* occurrence of the passed string.

Inserting/erasing

```
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& 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!*

Backwards compatibility

```
const char* string::c_str(void)
const char* string::data(void)
```

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 out;

      // Use the std::ostringstream just like std::cout or other iostreams
      out << "You have: " << 5 << " Helloworlds in your inbox";

      // Convert the std::ostringstream to a normal string
      std::string str = out.str();

      std::cout << str << std::endl;

      return 0;
}
```

Advanced use

TODO
Template parameters to basic_string etc.

Streams

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.

```
// 'Hello World!' program

#include <iostream>

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

demonstrates the use of the `std::cout` stream, known as the *standard output stream*.

IOStreams are part of the C++ Standard Library concept we saw early not to be confused with the Standard Template Library (STL) that we have yet to introduce.

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.

Stream classes

TODO
explain stream manipulators (left, right, ...), also mention buffers

Input and output is critical to computers. Every program you write will handle i/o in some form to communicate with the user. As this is a very common operation, programming languages like C++ are designed to make i/o as powerful yet painless as possible.

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 use, so C++, like other languages, provide an abstraction to make programming easier. This abstraction is the **stream**.

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.

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.

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.)

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

```
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.

Files

With `cout` and `cin`, we can do basic communication with the user. For more complex io, we would like to read to and write from files. This is done with a file stream, defined in the header **`<fstream>`**. **`ofstream`** is an output file stream, and **`ifstream`** is an input file stream.

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()
{
   fstream 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().

"safe bool" idiom

or Why define an **`operator void*`**`()` cast operator rather than an **`operator bool`**`()`?

This is so that we cannot write things like:

```
int foo = std::cin;
```

or, more importantly,

```
int bah;
std::cin << bah;    // observe: << instead of >>
```

by mistake. However, it is not perfect, as it allows other mistakes such as

```
delete std::cin;
```

though fortunately such errors are less likely, as **`delete`** should be used carefully in any case.

The state of the art would have us instead define a private nested class `dummy` within `std::ios`, and return a pointer-to-member-function of dummy -- hence allowing implicit conversion from that to **`bool`**, but not allowing many other operations. This is sometimes referred to as the "safe bool" idiom, and is motivated by the fact that C++'s **`bool`** type has implicit conversions both to and from **`int`** as a result of the standardization process.

Output

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, " << userName << 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
```

(The name will of course vary; we will discuss variables a little later.)

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, " << userName << 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.

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.

Making user-created classes compatible with the stream library

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;
 }
```

Exception Handling

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. Instead, 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 which is necessary for the handling code to decide in which way it should react on the exception.

Consider a code example for clarification:

```
void a_function()
{
   // This function does not return normally,
   // instead execution will resume at a catch block.
   // 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 another_function()
{
   // 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
      {
         a_function();
         // Because the function throws an exception,
         // the rest of the code in this block will not
         // be executed
      }
   catch(char const* p_string) // 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.
    }
 }
```

`try` and `catch` block combination

Partial handling

Consider the following case:

```
 void g()
 {
    throw "Exception";
 }

 void f() {
```

```
    int* i = new int(0);
    g();
    delete i;
 }
 int main() {
    f();
   return 0;
 }
```

Can you see the problem in this code ? If g throws an exception, the variable i is never deleted and we have a memory leak.

To prevent the memory leak, f() must catch the exception, and delete i. But f() can't handle the exception, it doesn't know how!

What is the solution then? f() shall catch the exception, and then rethrow it:

```
 void g()
 {
    throw "Exception";
 }

 void f() {
    int* i = new int(0)
    try
    {
       g();
    }
    catch (...)
    {
       delete i;
       throw; // This empty throw rethrows the exception we caught
                 // An empty throw can only exist in a catch block
    }
    delete i;
 }
 int main() {
    f();
    return 0;
 }
```

There's a better way though; see "Writing Exception-Safe Code" below for information on using "RAII" classes to avoid the need to write catch, which also explains why C++ can do better than "finally".

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 numerator, float denominator)
{
   if(denominator == 0.0)
      throw DivisionByZeroException();
```

```
}
enum MathOperators {DIVISION, PRODUCT};

float operate(int action, float argLeft, float argRight)
{
   if(action == DIVISION)
   {
      return divide(argLeft, argRight);
   }
   else if(action != PRODUCT))
   {
    // call the product function
    // ...
   }
   // No match for the action! action is an invalid agument
   throw InvalidArgumentException();
}

int main(int argc, char* argv[]) {
   try
   {
       operate(atoi(argv[0]), atof(argv[1]), atof(argv[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. An 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.

Throwing Objects

There are several ways to throw an exception object. Let's review them.

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. 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.

Stack unwinding

Consider the following code

```
void g()
{
 throw std::exception();
}

void f()
{
   std::string str = "Hello"; // This string is newly allocated
   g();
}

void 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.

Writing exception safe code

Guards

If you plan to use exceptions in your code (and you should), 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 *i = new int(2);
    *i = 3;
    g();
    // Oops, if an exception is thrown, i is never deleted
    // and we have a memory leak
    delete i;
 }

 int main()
 {
    try
    {
```

```
        f();
    }
    catch(...)
    { }
    return 0;
 }
```

Can you see the problem in this code? When an exception is thown, we will never run the line that deletes `i`!

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 class sucks! This type of class is best implemented using templates.
 class IntDeleter {
 public:
     IntDeleter(int* value)
     {
          m_value = value;
     }

     ~IntDeleter()
      {
          delete m_value;
     }

     // operator *, enables us to dereference the object and use it
     // like a regular pointer.
     int&  operator *()
     {
          return *m_value;
     }

 private:
      int *m_value;
 };
```

The new version of f():

```
void f()
 {
    IntDeleter i(new int(2));
    *i = 3;
    g();
    // No need to delete i, 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, like Java.

Note that the C++ Standard Library provides a templated guard by the name of auto_ptr.

Guide-lines

Because it is hard to write exception safe code, you should only use an exception when you have to - when an error has occurred which you can not handle. Do *not* use exceptions for the normal flow of the program. This example is *WRONG*.

```
void sum(int a, int b) {
      throw a+b;
 }
 int main() {
      int result;
      try
      {
         sum(2,3);
      }
      catch(int tmpResult)
      {
         // Here the exception is used instead of a return value!
         // This is  wrong!
         result = tmpResult;
      }
     return 0;
 }
```

Exceptions in 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.

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 initialisation 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 initialisation 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.

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 and operator overloading. The Standard Template Library (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, 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's 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 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>
```

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 `sets`, 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 numbers. 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 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's 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:

```
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.

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 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's 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 from 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;
    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.

Advantages and disadvantages

Some uses of templates, such as the `max()` function, were previously filled by function-like preprocessor macros.

```
// 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, 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" 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.

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's 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's 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's 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's no implementation of some_yet_other_function() in the object code, just as there's 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's 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' may prove enlightening. In Borland, supposedly, there's 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's 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.

Template Metaprogramming Overview

Template metaprogramming (TMP) refers to uses of the C++ template system to perform computation at compile-time within C++ 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.

Because template metaprogramming is something of an unintended use of C++'s template system, it is frequently somewhat cumbersome, though powerful. 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.

Traits classes could also be considered a primitive form of template metaprogramming; 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). More sophisticated TMP came later.

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 principal 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. TMP has since advanced considerably, and is now a practical tool for library builders in C++, though its complications mean that it is not generally appropriate for the majority of applications or systems programming contexts.

```
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 D {
public:
    D(void*);
};

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() {
          D<i> d = prim ? 1 : 0;
          a.f();
    }
};

template<>
class Prime_print<1> {   // full specialization to end the loop
   public:
      enum {prim=0};
      void f() {
            D<1> d = prim ? 1 : 0;
      };
};

#ifndef LAST
#define LAST 18
#endif

int main()
{
    Prime_print<LAST> a;
    a.f();
}
```

Example: Compile-time Factorial

Calculating factorials is naturally done recursively: 0! = 1, and for n>0, n! = n*(n-1)!. In C++ 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>::result` is given by `n*factorial<n-1>::result`:

```
template <unsigned n>
struct factorial
```

```
{
   enum { result = n * factorial<n-1>::result };
};
```

Next, the specialization for zero says that `factorial<0>::result` evaluates to 1:

```
template <>
struct factorial<0>
{
   enum { result = 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>::result ];
}
```

Example: Compile-time "If"

The following code defines a metafunction 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()
{
   if_<true, int, void*>::result number(3);
   if_<false, int, void*>::result pointer(&number);
}
```

TODO
Document the code above implementing if_

Building Blocks

TODO
Lay out some of the basic building blocks of TMP.

Conventions for "Structured" TMP

TODO
Describe some conventions for "structured" TMP.

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).

dynamic_cast

dynamic_cast allows *down-casts* of polymorphic types - in other words casting a base type to a type lower in the hierarchy.

```
object of target type = dynamic_cast<target type>(pointer expression);
object of target type = dynamic_cast<target type>(reference expression);
```

Let's say that we have the following class hierarchy:

```
class Interface
{
public:
      virtual void GenericOp() = 0;
};

class SpecificClass : public Interface
{
public:
      virtual void GenericOp();
      virtual void SpecificOp();
};
```

Let's say that we also have a pointer of type 'Interface', like so:

```
Interface* ptr_interface;
```

But suppose that we know *for sure* that this 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);
```

With this statement, 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.

typeid

The **typeid** operator returns information about a specific type.

```
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:

```
std::type_info info = typeid(ptr_interface);
std::cout << info.name() << std::endl;
```

This program would print 'SpecificClass' because that is the dynamic type of the pointer 'ptr_interface'.

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.

references to pointers wont 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.

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're doing.

Standard Template Library (STL)

C++'s Standard Library, incorporating much of the library known as the STL, makes programming easy. Instead of wondering if your array would ever need to hold 257 records or having nightmares of string buffer overflows, you can enjoy **string** and **vector** that automatically extend to contain more records.

The Standard Template Library (STL) offers easy-to-use containers, data types and functions. 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.

The true power of the STL lies not in its container classes, but in the fact that it is a framework, combining algorithms with data structures using indirection through iterators to allow generic implementations of algorithms to work efficiently on varied forms of data. To give a simple example, the same **std::copy** function in C++ 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>**. (For more details on std::vector, see #Containers.

```
// 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, a+10, b);

 // 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");
```

Containers

The conteiners we will discuss in this section of the book are part of the standard namespace (std::) and as such the class belongs to the **STL** (*Standard Template Libraries*).

NOTE:
When choosing a container, you should have in mind what makes them different, this will help you produce more efficient code.

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++.)

- 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 of elements, it can be used to create an array 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).

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).

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.
Watch out for how long that pointer is valid. You should also keep in mind that std::vector<T>::iterator may not be a pointer; the safer mode is to simply use the iterator. The contiguous nature of vectors is most often important when interfacing to C code.

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.

TODO
Add missing data

vector::Other Members

push_back - insert element at the end of the list in amortized constant time

pop_back - remove element from the end of the list in constant time

clear; a member function of the **vector** used to erase its data 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.

assign; a member function of the **vector** used to delete a *origin* **vector** and copies the specified elements to an empty *target* **vector**.

at; a member function of the **vector** that returns a reference to the data element at the specified location in the **vector**, with bounds checking

back; a member function of the **vector** that returns a reference to the last data element of the **vector**.

TODO
Add missing data

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;

        copy(istream_iterator<int>(cin), istream_iterator<int>(),
               back_inserter(vec));

        sort(vec.begin(), vec.end());

        cout << "Sorted: " << endl;

        copy(vec.begin(), vec.end(), ostream_iterator<int>(cout, "\n"));

        return 0;
 }
```

TODO
Rewriting to use the range constructor of `vector<int>` rather than using the copy algorithm introduces C++'s *most vexing parse*.

Linked lists

The STL provides a class template called **list** (part of the standard namespace (std::)) which implements a non-intrusive doubly-w:linked list. 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.

list examples

TODO
Add missing data

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

Iterators

C++'s iterators are the foundation of the STL, now largely incorporated into the standard library part of C++.

The basic idea of an iterator is to provide a way to navigate over some collection of objects. 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. Iterators are further divided based on properties such as traversal properties.

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, 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.

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
       )
 {
       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.

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.

```
/* INCORRECT Example demonstrating declaring a functor and using it
      to calculate the sum of squares of number in a vector */

  #include <vector>
  #include <alogrithm>

  // Define the Functor for AccumulateSquareValues
  template<typename T>
  struct AccumulateSquareValues
  {
        AccumulateSquareValues() : sumOfSquare()
        {
              sumOfSquare = 0;
        }
        bool operator()(const T& value)
        {
              sumOfSquare += value*value;
              // return false so that the looping in for_each continues
              return false;
        }
        T Result() const
        {
              return sumOfSquare;
```

```
        }
        T sumOfSquare;
};

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;
```

TODO: Fix the erroneous example above! Functors passed to for_each can be copied any number of times; therefore, they cannot accumulate state inside a member variable.

STL Algorithms

the STL algorithms are there mainly to help the programmer manipulate collections, sets or on elements in containers.

The _if suffix
The _copy suffix

- Non-modifying algorithms
- Modifying algorithms
- Removing algorithms
- Mutating algorithms
- Sorting algorithms
- Sorted range algorithms
- Numeric algorithms

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.

TODO
Could note that the **rebind** pattern used by allocators is an alternative to using a template template parameter. Historically the STL was largely developed before C++ compilers offered support for template template parameters. Interestingly, modern template metaprogramming style has promoted a rebind-like approach instead of using template template parameters.

Chapter Summary

1. I/O
    1. string
    2. Streams
2. Exception Handling
3. Templates
    1. Template Meta-Programming (TMP)
4. Run-Time Type Information (RTTI)
5. Standard Template Library (STL)

C++ Programming/Chapter Standard Library

Beyond the Standard (In the real world)

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, C++'s scoping mechanism can be used to control file access.

With RAII we can use for instance Class destructors to guarantee to clean up, similarly to functioning as *finally* keyword on other languages, doing 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. 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 or policy based `Loki::SmartPtr` from Loki library.

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 { } ;
 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.

Programming Patterns

Software design patterns are an emerging tool for guiding and documenting system design. A **pattern** is a way to define and wrap around a commonly accepted or identifiable name (most often named after a simplistic description of the goal it addresses), to a reusable/reproducible set of steps that address a specific goal, as a way to solve a generic programming problem (how generic or complex, depends on how restricted the target goal is). Patterns are commonly used in objected oriented programming languages like C++. This chapter explains when and how various patterns can be used.

A design pattern is not a finished design, it is a description of a solution to a common problem. A design pattern can be reused in multiple applications, and that is the main advantage of using it. It can also be see as a template for how to solve a problem that can occur in many different situations and/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 create 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.

A design pattern can be considered as block that can be placed in your design document, and you have to implement the design pattern with your application.

Using design patterns speeds up your design and helps to communicate your design to other team members.

Creational Patterns

In software engineering, **creational design patterns** are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

As we will see there are several creational design patterns, 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.

Factory Method

The Factory Design Pattern is useful in a situation where a design requires the creation of many different types of objects, all which come from a common base type. The Factory Method will take a description of a desired object at run time, and return a new instance of that object. The upshot of the pattern is that you can take a real word description of an object, like a string read from user input, pass it into the factory, and the factory will return a base class pointer. The pattern works best when a well designed interface is used for the base class, so that the returned object may be used fully without having to cast it, using RTTI.

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.

The following example uses the notion of laptop vs. desktop computer objects with a computer factory.

Let's start by defining out base class (interface) and the derived classes: Desktop and Laptop. Neither of these classes actually

"do" anything, but are meant for the illustrative purpose.

```
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
 };
```

Now for the actual Factory, 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;
     }
 };
```

Thanks to the magic of virtual functions, the caller of the factory method can use the returned results, without any knowledge of the true type of the object.

Let's analyze the benefits of this. 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. By doing this, that implementation file for NewComputer() is the only one which needs knowledge of the derived classes. Thus, if a change is made to any derived class of Computer, or a new Computer type is added, the implementation file for NewComputer() is the only file which needs to be recompiled. Everyone who uses the factory will only care about the interface, which will hopefully remain consistent throughout the life of the application.

Also, if a new class needs to be added, and the user is requesting objects through a user interface of some sort, no code calling the factory may need 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 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 description of levels, and developers could create new levels without any need to ever recompile their program. All this, thanks to the separation of creation of objects from the usage of objects.

Abstract Factory

Prototype Design Pattern [A fully initialized instance to be copied or cloned]

A prototype pattern is a creational design pattern 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.

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 );
     ~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("MyOwnMoster");
    // 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.

Prototype

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this 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 which is having pure virtual method "Clone()".
    - "CarRecord", "BikeRecord" and "PersonRecord" as concrete implementation of "Record" class.
    - An enum RECORD_TYPE_en as one to one mapping of each concrete implementation of "Record" class.
    - "RecordFactory" class which is having 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>

  using namespace std;

  enum RECORD_TYPE_en
  {
    CAR,
    BIKE,
    PERSON
  };

  /**
   * Record is the Prototype
   */

  class Record
  {
    public :

      Record() {}

      ~Record() {}

      Record* Clone()=0;

      virtual void Print()=0;
  };

  /**
   * CarRecord is Concrete Prototype
   */

  class CarRecord : public Record
  {
    private :
      string m_oStrCarName;

      u_int32_t m_ui32ID;

    public :

      CarRecord(string _oStrCarName,u_int32_t _ui32ID)
        : Record(), m_oStrCarName(_oStrCarName),
          m_ui32ID(_ui32ID)
      {
      }

      CarRecord(CarRecord& _oCarRecord)
        : Record()
      {
        m_oStrCarName = _oCarRecord.m_oStrCarName;
        m_ui32ID = _oCarRecord.m_ui32ID;
      }

      ~CarRecord() {}

      Record* Clone()
      {
        return new CarRecord(*this);
      }
```

```
    void Print()
    {
      cout << "Car Record" << endl
        << "Name  : " << m_oStrCarName << endl
        << "Number: " << m_ui32ID << endl << endl;
    }
};


/**
 * BikeRecord is the Concrete Prototype
 */

class BikeRecord : public Record
{
  private :
    string m_oStrBikeName;

    u_int32_t m_ui32ID;

  public :
    BikeRecord(string _oStrBikeName,u_int32_t _ui32ID)
      : Record(), m_oStrBikeName(_oStrBikeName),
        m_ui32ID(_ui32ID)
    {
    }

    BikeRecord(BikeRecord& _oBikeRecord)
      : Record()
    {
      m_oStrBikeName = _oBikeRecord.m_oStrBikeName;
      m_ui32ID = _oBikeRecord.m_ui32ID;
    }

    ~BikeRecord() {}

    Record* Clone()
    {
      return new BikeRecord(*this);
    }

    void Print()
    {
      cout << "Bike Record" << endl
        << "Name  : " << m_oStrBikeName << endl
        << "Number: " << m_ui32ID << endl << endl;
    }
};


/**
 * PersonRecord is the Concrete Prototype
 */

class PersonRecord : public Record
{
  private :
    string m_oStrPersonName;

    u_int32_t m_ui32Age;

  public :
    PersonRecord(string _oStrPersonName, u_int32_t _ui32Age)
      : Record(), m_oStrPersonName(_oStrPersonName),
        m_ui32Age(_ui32Age)
    {
    }
```

```
    PersonRecord(PersonRecord& _oPersonRecord)
      : Record()
    {
      m_oStrPersonName = _oPersonRecord.m_oStrPersonName;
      m_ui32Age = _oPersonRecord.m_ui32Age;
    }

    ~PersonRecord() {}

    Record* Clone()
    {
      return new PersonRecord(*this);
  }

  void Print()
  {
    cout << "Person Record" << endl
      << "Name : " << m_oStrPersonName << endl
      << "Age  : " << m_ui32Age << endl << endl ;
  }
};


/**
 * RecordFactory is the client
 */

class RecordFactory
{
  private :
    map<RECORD_TYPE_en, Record* > m_oMapRecordReference;

  public :
    RecordFactory()
    {
      m_oMapRecordReference[CAR]  = new CarRecord("Ferrari", 5050);
      m_oMapRecordReference[BIKE] = new BikeRecord("Yamaha", 2525);
    m_oMapRecordReference[PERSON] = new PersonRecord("Tom", 25);
    }

    ~RecordFactory()
    {
      delete m_oMapRecordReference[CAR];
      delete m_oMapRecordReference[BIKE];
      delete m_oMapRecordReference[PERSON];
    }

    Record* CreateRecord(RECORD_TYPE_en enType)
    {
      return m_oMapRecordReference[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;
  }
```

Singleton

This section assumes previous familiarity with Functions, Global Variables, Stack vs. Heap, Classes, Pointers, and static Member Functions.

TODO
Link to sections that would describe these.

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.

TODO
Other pros/cons of the use of singletons.

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:

TODO
Discussion of Meyers Singleton and any other variations.

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 to use globals liberally, pulling these into a single class and making it a singleton, can allow an intermediary step to bringing a structural program into an object oriented structure.

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.

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.

Composite

Components that are individual objects and also can be collection of objects. A Composite pattern can represent both the conditions. In this pattern, one can develop tree structures for representing part-whole hierarchies.

Decorator

The decorator pattern helps to add behavior or responsibilities to an object. This is also called “Wrapper”.

Facade

A Facade pattern hides the complexities of the system and provides an interface to the client from where the client can access the system.

Flyweight

It is a mechanism by which you can avoid creating a large number of object instances to represent the entire system. 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

It is used when you need to represent a complex with a simpler one. If creation of object is expensive, its creation can be postponed till the very need arises and till then, a simple object can represent it. This simple 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.

Behavioral Patterns

Chain of Responsibility

Command

Command pattern is an Object behavioral pattern that decouples sender and receiver. 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.

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's 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;
 }
 // Don't 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:

TODO
test compiling this, check for syntax errors.

```
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_back();
     } // 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;
     }
     // Add this value to the end 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 dont 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 easier the usage of the container easier, and hides the traversal logic from other developers.

Let's 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;
```

TODO

- Discussion of iterators in the STL, and the usefulness of iterators within the algorithms library.
- Iterators best practices
- Warnings on creation of and usage of

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

Helps in implementing objects state without breaking encapsulation. Though GoF uses friend to implement this but it is not a good design. It can be implemented using PIMPL. Best Use case is 'Undo-Redo' in an editor.

Observer

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 unsubscribed.

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.

State

Strategy

Template Method

Visitor

Model-View-Controller (MVC)

pattern often used by applications that need the ability to maintain multiple views of the same data.

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 specific functions that the library provides 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* or API. A more indent examination is provided in the APIs and *Frameworks* Section of this book.

As seen in the File Organization Section, compiled libraries consists of two parts:

- **H/HPP** (header) files, which contains the interface definitions
- **LIB** (library) files, which contains the library itself

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 used by third party libraries, the Static and Dynamic Libraries Section of this book will cover in depth this subject.

Additional functionality that goes beyond the standard libraries (like garbage collection) are available for C++ by (often free) third party libraries, but remember that third party libraries do not provide the same ubiquitous cross-platform functionality or an API style conformant with as standard libraries. Exceptions are rare one of the most clean example is the highly respected Boost Library that we will examine ahead.

Here, we will try to include all major libraries that the programmer should know about or have at least a passing idea of what they are, there is a clear difficulty on knowing all existing libraries available as the number is always increasing. This third party libraries can be looked as extensions to the standard or just as a life line for surviving on an alien OS API. 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.

APIs and frameworks

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 (ie: 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, it is a set of solutions or even classes (in case of some *C++ frameworks*) that in group addresses the handling of a limited set of related problems and provides not only an API but a default functionality that can be replaced by a similar *framework*, even better if it provides the same API.

Static and Dynamic Libraries

Binary/Source Compatibility

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 needs to be recompiled to run with a new version of library but doesn't require any further modifications, the library is *source compatible*. If a program does not need to be modified and recompiled in order to use a new version of a library, the library is *binary compatible*.

Binary compatibility saves a lot of trouble. It makes it much easier to distribute software for a certain platform. Without ensuring binary compatibility between releases, people will be forced to provide statically linked binaries. By linking dynamically to library (even a former version of the library), it will continue running with newer versions of the library without the need to recompile.

Using static binaries is bad because they:

- waste resources (especially memory but this may depend on the OS)
- don't allow the program to benefit from bug fixes or extensions in the libraries

Using static binaries is good because:

- will simplify (take less files for) the distribution
- will simplify the code (ie: no version checks)

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 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.

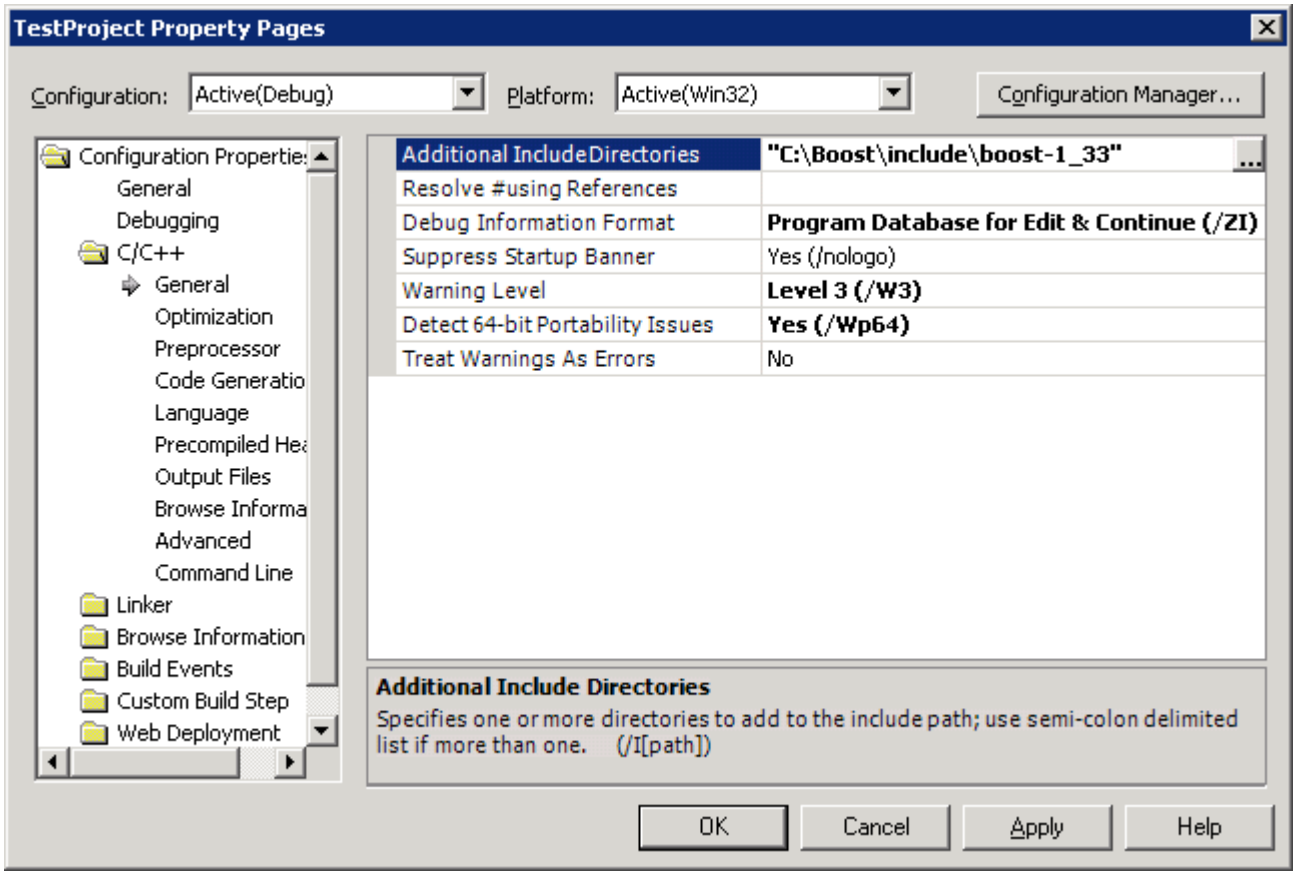
TODO
Complete with missing info

Procedure to use static libraries (Visual Studio)

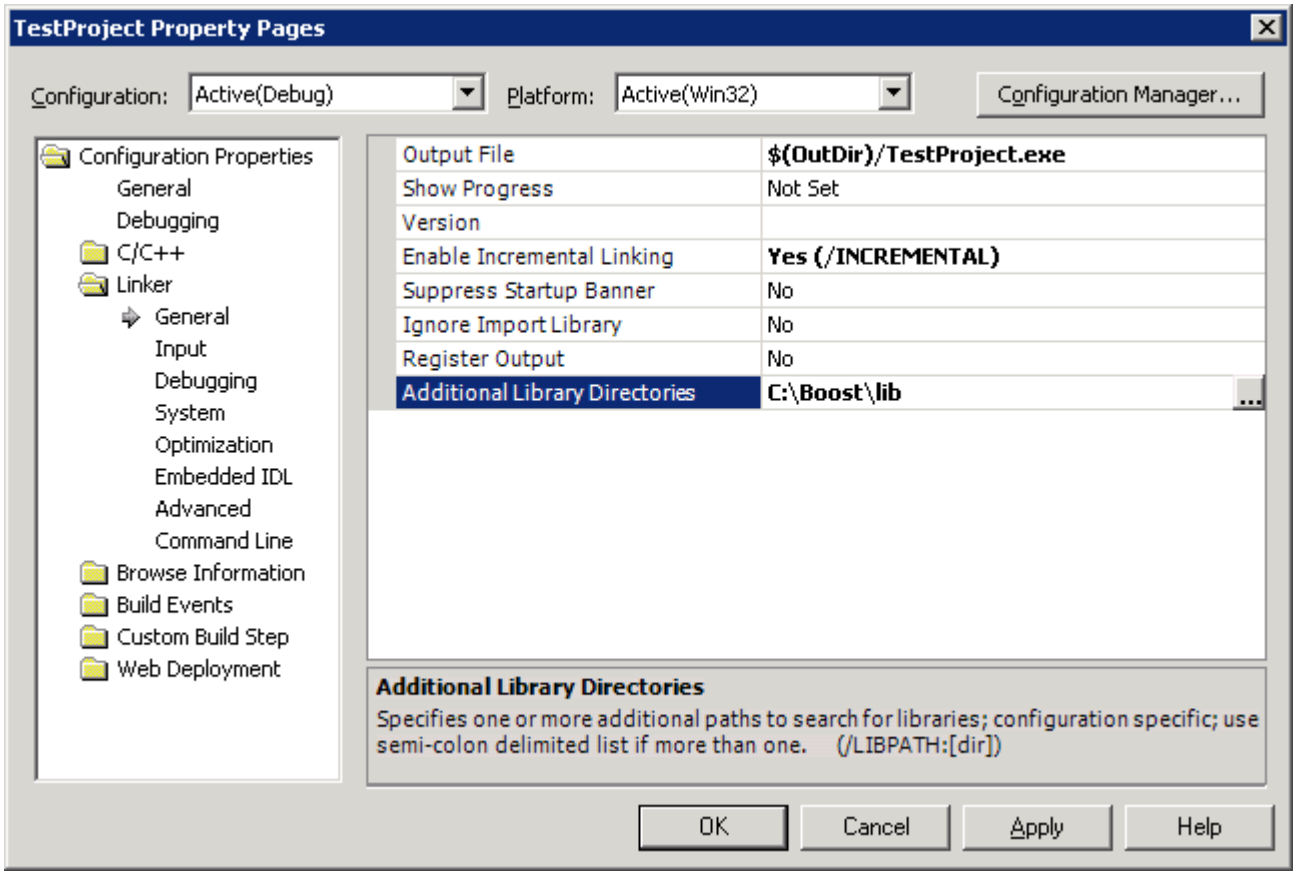
This section serves an an example on how to set up static libraries for usage in Visual Studio. The Boost library serves as an example library.

There are three steps which have to be performed:

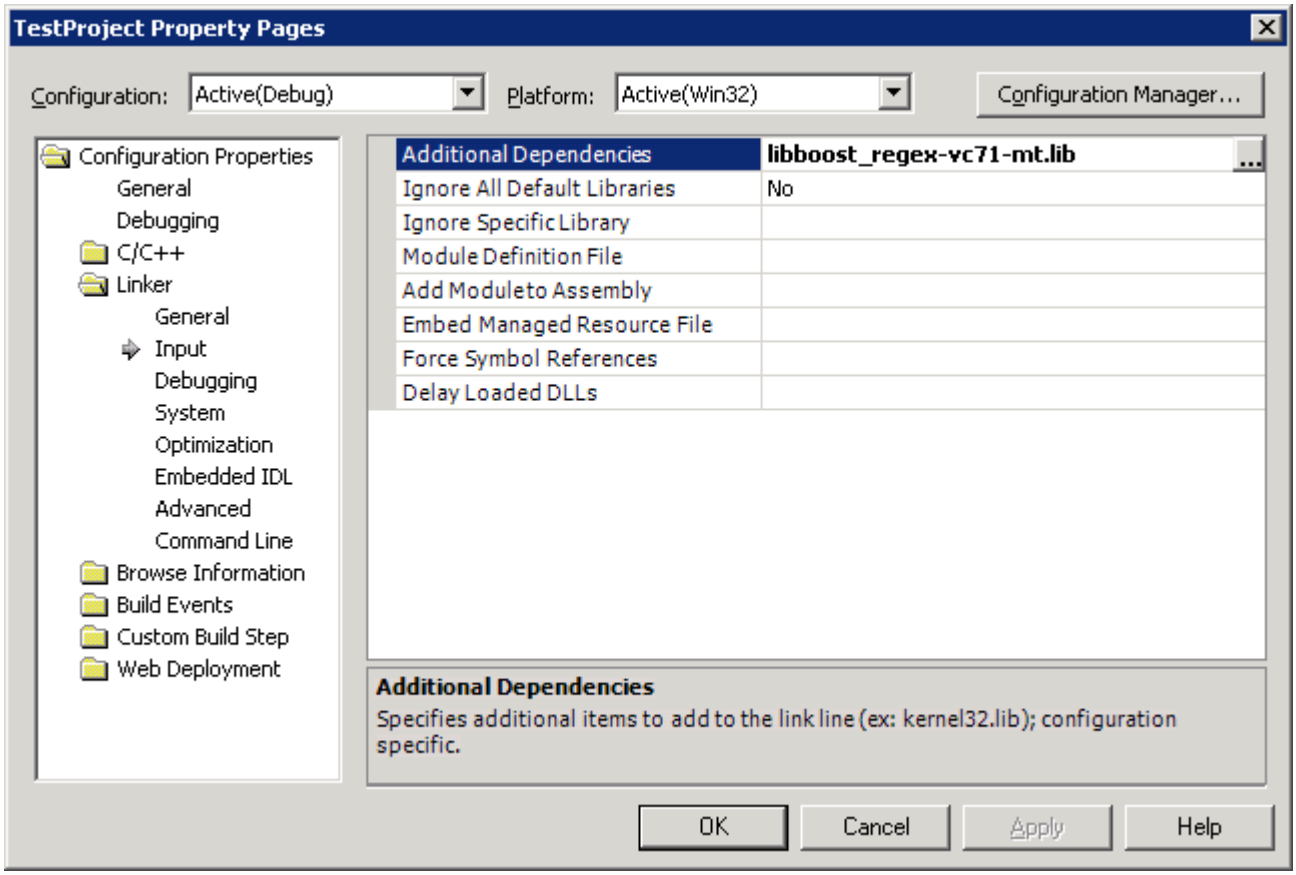
- Set up the *include directory*, which contains the header files of your library



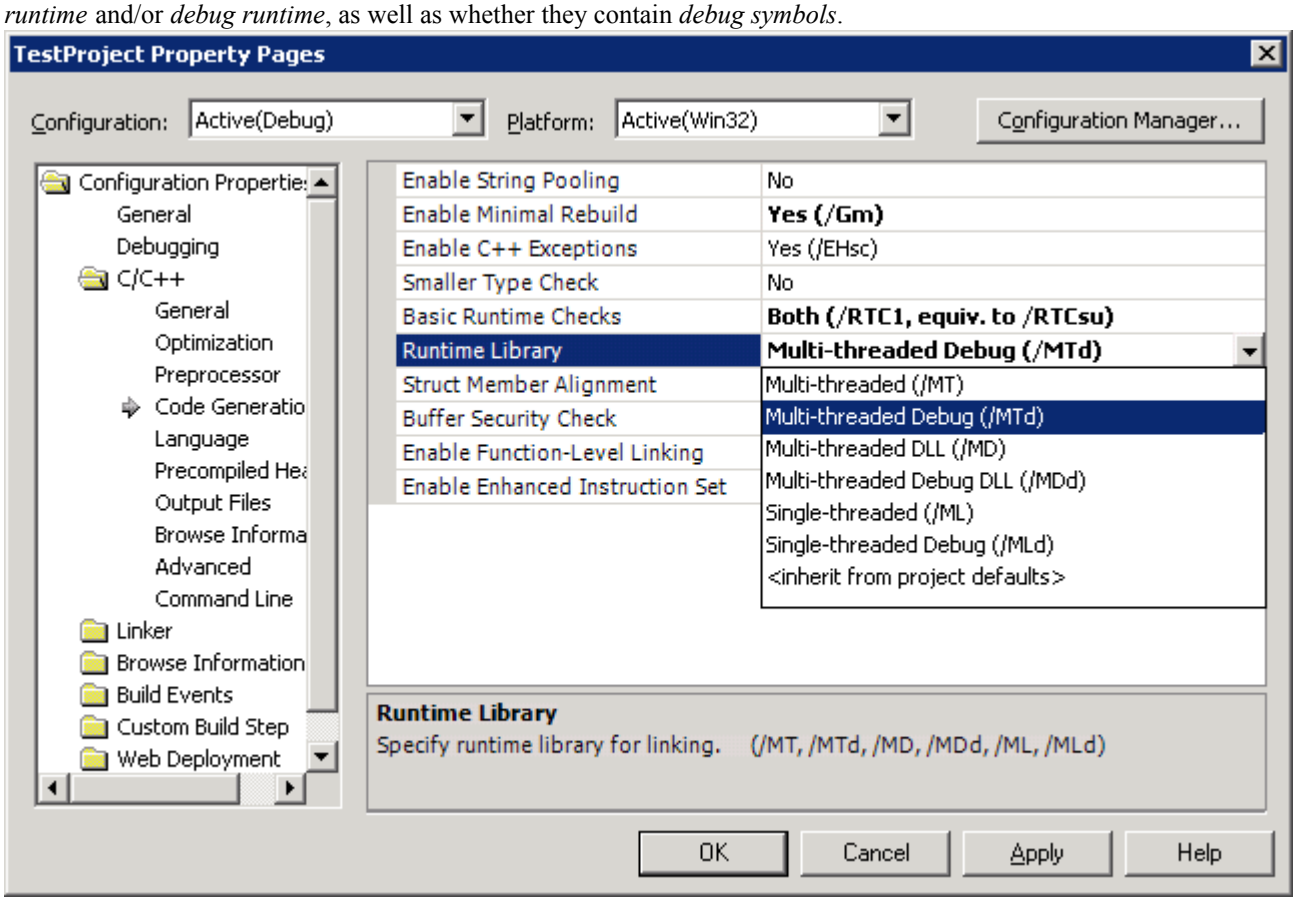
- Set up the *library directory*, which contains the path of the library file(s)



- Enter the library filename(s) in *additional dependencies*



The libraries selected here have to be compiled for the same run-time library as the one used in your project. Most static libraries does therefore come in different editions, depending on whether they are compiled for *single- or multithreaded runtime* and/or *debug runtime*, as well as whether they contain *debug symbols*.



More specific information on about Boost can be found on the Boost Library Section of this book.

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 GC 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 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. C++ also supports a powerful idiom called *RAII (resource acquisition is initialization)* that is covered in the RAII Section of this book that states that RAII can be used to safely and automatically manage resources including memory.

TODO
Add some examples into libraries and extend a bit more this info

Boost Library

The **Boost library** (http://www.boost.org/) provides free peer-reviewed , open source libraries that extend the functionality of C++. Most of the libraries are licensed under the Boost Software License, designed to allow Boost to be used with both open and closed source projects.

Many of Boost's founders are on the C++ standard committee and several Boost libraries have been accepted for incorporation into the Technical Report 1 of C++0x. 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 in regular use by thousands of programmers. They range from general-purpose libraries like SmartPtr, to OS Abstractions like FileSystem, to libraries primarily aimed at other library developers and advanced C++ users, like MPL.

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 upcoming C++ Standard Library Technical Report as a step toward becoming part of a future C++ Standard.

In order to ensure efficiency and flexibility, Boost makes extensive use of templates. Boost has been a source of extensive work and research into generic programming and metaprogramming in C++.

extension libraries

- Algorithms
- Concurrent programming (threads)
- Containers
    - array - Management of fixed-size arrays with STL container semantics
    - Boost Graph Library (BGL) - Generic graph containers, components and algorithms
    - multi-array - Simplifies creation of N-dimensional arrays
    - multi-index containers - Containers with built in indexes that allow different sorting and access semantics
    - pointer containers - Containers modeled after most standard STL containers that allow for transparent management of pointers to values
    - property map - Interface specifications in the form of concepts and a general purpose interface for mapping key values to objects
    - variant - 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
    - concept check - Allows for the enforcement of actual template parameter requirements (concepts)
    - static assert - Compile time assertion support
    - Boost Test Library - 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 - Dynamic `std::bitset`-like data structure
- Function objects and higher-order programming
    - bind and mem_fn - General binders for functions, function objects, function pointers and member functions
    - function - Function object wrappers for deferred calls. Also, provides a generalized mechanism for callbacks
    - functional - Enhancements to the function object adapters specified in the C++ Standard Library, including:
        - function object traits
        - negators
        - binders
        - adapters for pointers to functions
        - adapters for pointers to member functions
    - hash - 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 - In the spirit of lambda abstractions, 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 - Provides utility class templates for enhancing the capabilities of standard C++ references, especially for use with generic functions
    - result_of - Helps in the determination of the type of a call expression
    - signals and slots - Managed signals and slots callback implementation
- Generic programming
- Graphs
- Input/output
- Interlanguage support (for Python)
- Iterators
    - iterators
    - operators - Class templates that help with overloaded operator definitions for user defined iterators and classes that can participate in arithmetic computation.
    - tokenizer - 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
  - pool - Provides a simple segregated storage based memory management scheme
  - smart_ptr - A collection of smart pointer class templates with different pointee management semantics
    - scoped_ptr - Owns the pointee (single object)
    - scoped_array - Like scoped_ptr, but for arrays
    - shared_ptr - Potentially shares the pointer with other shared_ptrs. Pointee is destroyed when last shared_ptr to it is destroyed
    - shared_array - Like shared_ptr, but for arrays
    - weak_ptr - Provides a "weak" reference to an object that is already managed by a shared_ptr
    - intrusive_ptr - Similared to shared_ptr, but uses a reference count provided by the pointee
  - utility - Miscellaneous support classes, including:
    - base from member idiom - 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 - 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 - 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 - Allows for the prohibition of copy construction and copy assignment
    - addressof - Allows for the acquisition of an object's real address, bypassing any overloads of operator&(), in the process
    - result_of - Helps in the determination of the type of a call expression
- Miscellaneous
- Parsers
- Preprocessor metaprogramming
- String and text processing
  - lexical_cast - Type conversions to/from text
  - format - Type safe argument formatting according to a format string
  - iostreams - C++ streams and stream buffer assistance for new sources/sinks, filters framework
  - regex - Support for regular expressions
  - Spirit - An object-oriented recursive-descent parser generator framework
  - string algorithms - A collection of various algorithms related to strings
  - tokenizer - Allows for the partitioning of a string or other character sequence into tokens
  - wave - Standards conformant implementation of the mandated C99 / C++ pre-processor functionality packed behind an easy to use interface
- Template metaprogramming
  - mpl - A general purpose high-level metaprogramming framework of compile-time algorithms, sequences and metafunctions
  - static assert - Compile time assertion support
  - type traits - Templates that define the fundamental properties of types
- Workarounds for broken compilers

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.

Darwin and Mac OS X

Both Mac OS X version 10.x.x and "pure" Darwin systems can be to as Darwin, since Mac OS X is actually a superset of Darwin.

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 interfaces available in the Microsoft Windows operating systems. It is designed for usage by C/C++ programs and is the most direct way to interact with a *Windows* system for software applications. Lower level access to a *Windows* system, mostly required for device drivers, is provided by the Windows Driver Model 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/) 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.

A software development kit (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/)

See Also: Windows Programming

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.

Charles Petzold, 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 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 supported less then 450 function calls, 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 on maintaining software backwards compatibility. To achieve this, Microsoft sometimes went as far as supporting software that was using the API in a undocumented or even (programmatically) illegal way. Raymond Chen, 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 standart 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.

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 Common Object Model implemented as part of Windows.

TODO

Add **Registry, IO (Input/Output), Security, Processes and Threads**

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=/library/en-us/sysinfo/base/time_functions.asp)

Timer Function Performance (http://developer.nvidia.com/object/timer_function_performance.html)

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.

Performance counter value may unexpectedly leap forward (http://support.microsoft.com/default.aspx?scid=KB;EN-US;Q274323&)

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.

TODO

Add **Basics in building "windows", Window eventhandling, Resources**

Resources

Resources are perhaps one of the most useful elements of the WIN32 API, they are how we program menu's, add icons, backgrounds, music and many more aesthetically pleasing elements to our programs.

They 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:

```
////////////// 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
```

Network

Network applications are often built in C++ on windows utilizing the WinSock API functions.

WIN32 API Wrappers

Microsoft Foundation Classes (**MFC**);

> a C++ library for developing Windows applications and UI components. Created by Microsoft for the C++ Window's 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). More info about **MFC** can be obtained at http://en.wikibooks.org/wiki/Windows_Programming#Section_3:_Microsoft_Foundation_Classes_and_COM

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/) a library of C++ classes that extend Microsoft Foundation Classes (MFC) to do NT specific things.

Borland Visual Components Library (**VCL**);

a Delphy/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 nonvisual, and non windows-specific objects, like AnsiString class for example.

Generic wrappers

Generic GUI/API wrappers are programming libraries that provide an uniform platform neutral interface (API) to the operating system regardless of underlying platform. Such libraries greatly simplify development of cross-platform software.

- **Gtkmm** - an interface for the GUI library GTK+
- **Qt** - a cross-platform graphical widget toolkit for the development of GUI programs
- **WxWidgets** (http://www.wxwindows.org/) - a framework that lets developers create applications for 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.

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.

Multitasking

Multitasking is a method by which multiple tasks, also known as processes, share common processing resources such as a CPU. A computer with a single CPU, will only run one task, *running* means that in a specific point in time, the CPU is actively executing instructions for that process.

A single CPU systems using scheduling can archive multitasking by which a task may be run at any given time, and then another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called a context switch. When context switches occur frequently enough the illusion of parallelism is achieved. Even on computers with more than one CPU (called multiprocessor machines), multitasking allows many more tasks to be run than there are CPUs.

Operating systems may adopt one of many different scheduling strategies, which generally fall into the following categories:

- In *multiprogramming* 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* systems, the running task is required to relinquish the CPU, either voluntarily or by an external event such as a hardware interrupt. 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 *multitasking*.

- In *real-time* 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.

Processes

Processes are independent execution units that contain their own state information, use their own address spaces, and only interact with each other via interprocess communication mechanisms (IPC).

Child Process

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 need of enabling random-access to the shared data is addressed. By declaring a given section of memory that can be used simultaneously by several processes. Enabling this memory segment to be 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.

Threads

Threads are by definition a coding construct and part of a program that enable it to fork (or split) itself into two or more simultaneously (or pseudo-simultaneously) running tasks.

Threads vs. Processes

Both threads and processes are methods of parallelizing an application, its implementation may differ from one operating system to another. In general, a thread is contained inside a process and different threads of the same process share some resources while different processes do not.

Cooperative vs. Preemptive Threading

Multi Threading

Unlike more modern languages like Java and C#, the C++ standard does not include specifications or built in support for multi-threading. Threading must therefore be implemented using special threading libraries, which are often platform dependent.

Some popular threads libraries for C++ include

- Boost - 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.
- Zthreads [1] - Another popular, portable thread abstraction library. This library is more feature rich, and deals only with concurrency.
- ACE [2] - Another toolkit which includes a portable threads abstraction along with many many other facilities, all rolled into one library.

This list is not intended to be complete.

Of course, you can access the full POSIX and Windows C language threads interface from C++. 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.

Example

The CreateThread function creates a new thread for a process. The creating thread must specify the starting address of the code that the new thread is to execute. Typically, the starting address is the name of a function defined in the program code. This function takes a single parameter and returns a DWORD value. A process can have multiple threads simultaneously executing the same function.

The following example demonstrates how to create a new thread that executes the locally defined function, ThreadFunc.

```
DWORD WINAPI ThreadFunc( LPVOID lpParam )
{
   char szMsg[80];
   wsprintf( szMsg, "ThreadFunc: Parameter = %d\n", *lpParam );
   MessageBox( NULL, szMsg, "Thread created.", MB_OK );
   return 0;
}
VOID main( VOID )
{
   DWORD dwThreadId, dwThrdParam = 1;
   HANDLE hThread;
   hThread = CreateThread(
       NULL,                        // no security attributes
       0,                           // use default stack size
       ThreadFunc,                  // thread function
       &dwThrdParam,                // argument to thread function
       0,                           // use default creation flags
       &dwThreadId);                // returns the thread identifier

  // Check the return value for success.
  if (hThread == NULL)
     ErrorExit( "CreateThread failed." );
  CloseHandle( hThread );
}
```

For simplicity, this example passes a pointer to a DWORD value as an argument to the thread function. This could be a pointer to any type of data or structure, or it could be omitted altogether by passing a NULL pointer and deleting the references to the parameter in ThreadFunc. It is risky to pass the address of a local variable if the creating thread exits before the new thread, because the pointer becomes invalid. Instead, either pass a pointer to dynamically allocated memory or make the creating thread wait for the new thread to terminate. Data can also be passed from the creating thread to the new thread using global variables. With global variables, it is usually necessary to synchronize access by multiple threads.

Thread Local Variables
Thread Synchronization
Synchronizing on Objects
Suspend and Resume
Deadlock

Optimizing Your Programs

Optimization can be regarded as a directed effort to increase the performance of something, in the particular case we will cover, we will deal with specific computational tasks and best practices to reduce resources utilizations not only the system resources but programmers and users. This tasks often a topic of discussion among programmers and not all conclusion may be consensual and may depend on the goals, experience and some claims can only be substantiated by doing a profiling the given problem.

The level of optimization depends directly from actions and decisions the programmer makes. It can depend on simple things, like the selection of the tools one choses to use to create the program, even selecting the compiler has some impact, to simple code practices that can reduce the size or increase the speed on the final product. All optimization step taken should have as a goal the reduction of requirements and the promotion of the program objectives.

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 of the book.

Memory footprint

In the past computer memory has been expensive and technologically limited in size, a scarce resource for programmers, a 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 a washing machine has enough capacity to simulate a supernova explosion but has capacity augments the needs and expectations have also evolved.

TODO
Continue

Wikipedia has related information at ***Unified Modeling Language***.

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.

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 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 (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, OOSE, Class-Relation and OOramand 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 it's adoption in November 1997 by the OMG (Object Management Group) and it's 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.

Chapter Summary

1. Resource Acquisition Is Initialization (RAII)
2. Design Patterns (Creational, Structural and Behavioral patterns)
3. Libraries
    1. APIs and *Frameworks*
    2. Static and Dynamic Libraries
    3. Garbage Collection
    4. Boost Library
4. Cross-Platform Development
    1. Win32 (aka WinAPI) - including Win32 Wrappers.
    2. Cross Platform Wrappers
    3. Multitasking
5. Optimizing Your Programs
6. Unified Modeling Language (UML)

Appendex 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.

Appendix B: External References

External links

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

  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/ , 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.

Compilers and IDEs

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 very good Win32 port of the GNU Compiler Collection and toolset.

- http://sourceware.cygnus.com/cygwin

  Cygwin, another free and good Win32 port of GCC and GNU Utils.

- http://msdn.microsoft.com/vstudio/express/visualC/default.aspx

  The *Microsoft Visual C++ 2005 Express Edition*. It also allows you to build applications that target the Common Language Runtime (CLR). You should read their licence 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

(http://www.microsoft.com/msdownload/platformsdk/sdkupdate/) 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 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.codeblocks.org/

  Code::Blocks, C++ cross-platform and Open Source (GPL2) IDE, runs on Linux or Windows (uses wxWidgets), suports 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.

- 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

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. You can purchase Comeau C/C++ 4.3.3 for $50. Core C++03 language enhancements for all major and minor features of C++ and C, including export.

Libraries

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/

  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, 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, 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 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 framebuffer 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 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.

General Information/Discussion Forums

References to other locations/works or discussion areas that can be relevant to the topic of the book.

Misc.

- http://del.icio.us/tag/"c++"

  A community Bookmark ranking/sharing WEB tool lots of related sites to the C++ topic.

- http://www.cppreference.com/

  Kohl et al. 2004: C/C++ Reference

- http://www.icce.rug.nl/documents/cplusplus/

  The C++ Annotations for C programmers

- http://www.cplusplus.com/main.html

  cplusplus.com is an open resource for visitors. Here you will find different web discussion groups where ask what you always wanted to know, share experiences and discoveries and help other programmers.

- http://www.aristeia.com/

  Scott Meyers is one of the world's foremost experts on C++ software development. He wrote the best-selling Effective C++ series (Effective C++, More Effective C++, and Effective STL), wrote and designed the innovative Effective C++ CD, is consulting editor for Addison Wesley's Effective Software Development Series, and is a member of the advisory board for Software Development magazine.

- http://www.cprogramming.com/

  Cprogramming.com is a web site designed to help you learn C or C++ and provide you with C and C++ programming resources.

- http://cprog.oreilly.com/

  Misc. Books, News & Articles on C++.

- http://www.iseran.com/Win32/FAQ/

  Frequently Asked Questions About Win32 Programming.

- http://c2.com/cgi/wiki?CppHeresy

  CppHeresy, provides guidelines about how to keep the C++ bits simple.

- http://www.oonumerics.org/mailman/listinfo.cgi/oon-list/

  Object-Oriented Numerics List is a forum for discussing scientific computing in object-oriented environments. An archive is available.

News and Publications

- http://www.cuj.com/

  C/C++ Users Journal.

IRC

- #C at (irc://irc.tambov.ru)

  C/C++ Channel (http://silversoft.net/) (Russian Channel)

- #C++ at (irc://hub.ptnet.org)

  C++ Channel (Portuguese Channel)

- ##C++ at (irc://irc.freenode.net)

  C++ Channel

- #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.

User Groups

- http://www.accu.org/

  ACCU, formerly the Association for C and C++ Users, ACCU is a non-profit organisation devoted to professionalism in programming at all levels. Although primarily focussed on C and C++, have now interests in Java, C# and Python also.

Newsgroups (NNTP)

- comp.std.c++ - FAQ
- comp.lang.c++.leda
- comp.lang.c++.moderated
- comp.lang.c++
- microsoft.public.vc.mfc
- microsoft.public.vc.stl

Blogs and Wikis

- http://www.codepedia.com/1/Cpp

  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).

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.

Forums

- http://invisionfree.com/forums/CPPlearningcommunity/

  C++ Learning Community, forum to discuss C++ related topics. Beginners made especially welcomed.

- http://www.nystic.com

  New to the C++ world? Go and ask any questions you may have.

Online Books

- http://en.wikibooks.org/wiki/Windows Programming

  Windows API (C and VB Classic), MFC (C++), COM and creation of ActiveX modules.

- http://www.relisoft.com/book/index.htm

  C++ In Action

- http://guides.oernii.sk/c++/index.htm

  Teach Yourself C++ in 21 Days, Second Edition

- http://www-h.eng.cam.ac.uk/help/tpl/languages/C++/doc/doc.html

  More C++ Tim Love July 5, 2001

- http://www.zib.de/Visual/people/mueller/Course/Tutorial/tutorial.html

  Introduction to Object-Oriented Programming Using C++, Peter Müller 1997

- http://math.nist.gov/~RPozo/c++class/

  C++ Programming for Scientists,

  Roldan Pozo, Computing and Applied Mathematics Laboratory, Karin Remington, Scientific Computing Environments Division

- http://www.cs.jcu.edu.au/~david/C++SYNTAX.html

  C++ for Unix quick-reference, with C variations

- http://www.halpernwightsoftware.com/stdlib-scratch/quickref.html

  STL Quick Reference

- http://www.steveheller.com/cppad/Output/dialogTOC.html

  C++ A dialog - Steve Heller

- http://cs.nmhu.edu/personal/curtis/cs1htmlfiles/CS1TEXT6-2001.HTM

  Learning C++: An Index of Entry Points

- http://www.digilife.be/quickreferences/Books/C++%20Programming%20HOW-TO.pdf

  C++ Programming How-To

- http://ibiblio.org/obp/thinkCS/cpp.php

  Think like a computer scientist: C++

- http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html

  Thinking in C++, 2nd Edition by Bruce Eckel, Free Electronic Book, Volume 1 & Volume 2

Misc. C++ Tools

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 optimising 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)

C++ Coding Conventions

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

- http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

  Java style

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

Other (dead tree) books on C++

Introductory books

- Thinking in C++, Volume 1: Introduction to Standard C++ by Bruce Eckel, ISBN 0139798099. (available for free download.)

Advanced topics

- Thinking in C++, Volume 2: Practical Programming by Bruce Eckel, ISBN 0130353132
- Effective C++ : 55 Specific Ways to Improve Your Programs and Designs, 3rd ed. by Scott Meyers, ISBN 0321334876

Reference books

- C++ in a Nutshell by Ray Lischner, ISBN 059600298X
- C++ Pocket Reference by Kyle Loudon, ISBN 0596004966
- C++ FAQs by Marshall Cline, Greg Lomow, and Mike Girou, Addison-Wesley, 1999, ISBN 0-201-30983-1

GNU Free Documentation License

Version 1.2, November 2002

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.

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 for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

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.