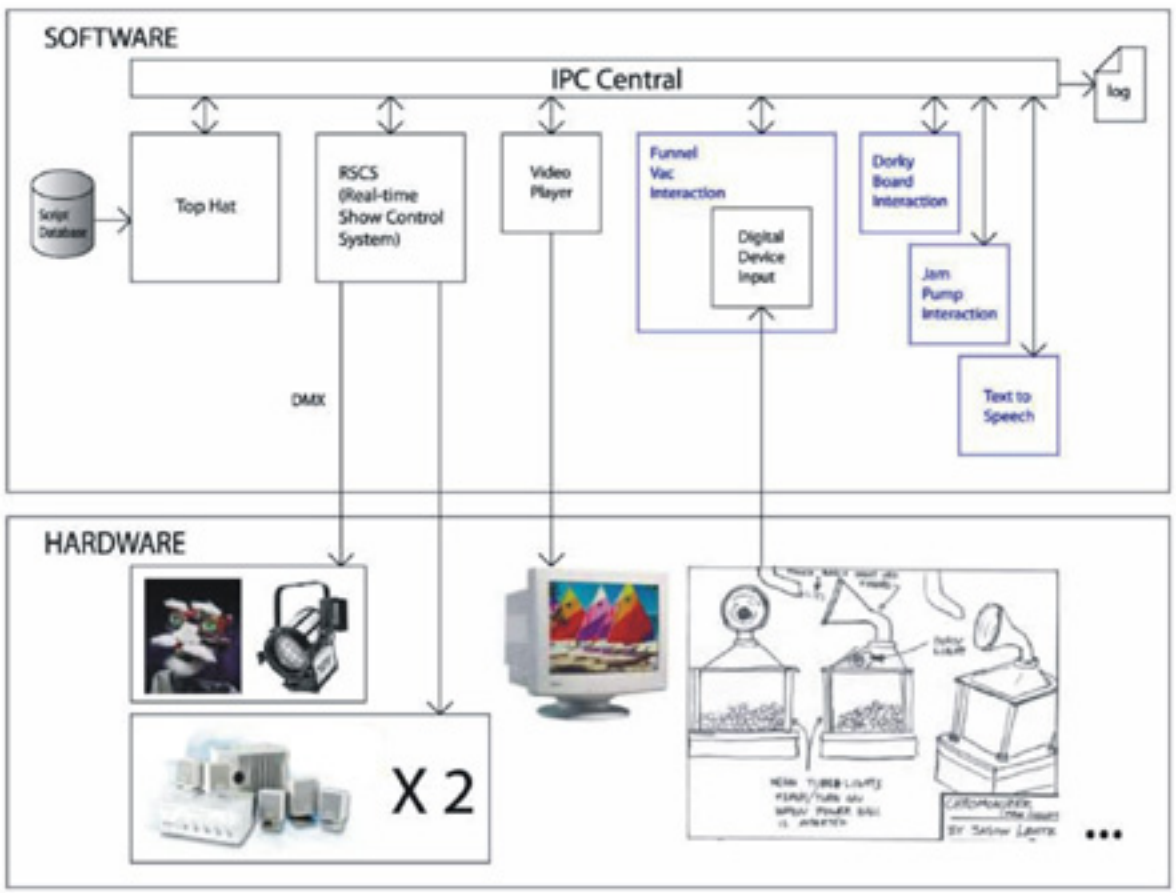
C++ Programing

Donald Walker



Training makes a difference...

Global Media
Education For Everyone

First Edition, 2007

ISBN 978 81 89940 37 9

Published by:

Global Media
1819, Bhagirath Palace,
Chandni Chowk, Delhi-110 006
Email: globalmedia@dkpd.com

Table of Contents

Introduction to C++

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

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, template, and exception handling. The C++ programming language standard was ratified in 1998 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 C's Standard Library. 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.

The C++ language is not "a better C", it is mainly an extension of C dedicated to extend it into the generic and object-oriented programming paradigm. 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, and run-time type identification.

C++ also performs more type checking than C in several cases.

Comments starting with two slashes ("`//`") were originally part of C's predecessor, BCPL, and were reintroduced in C++.

Several features of C++ were later adopted by C, including `const`, `inline`, declarations in `for` loops, and C++-style comments (using the `//` symbol).

C++ source code example

```
// 'Hello World!' program

#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
    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.

Information for the reader

If you already know how to program in any other language you should best skip to the section *Understanding Source Files* in the *Getting Started* chapter. You should check also the next section so you can get a feel for what is different from the language you are accustomed to.

If this is your first contact with a programming language then continue on reading, 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 concept 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.

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 no one can "speak" long lines of binary... the complexity of writing large systems is enormous with high-level languages, and beyond human capabilities with machine code. Below is a quick comparison between C++ and other programming languages.

C 89/99

C is the predecessor of C++ and the sequel to Basic ("B"). C is essentially the core language of C++, and many of the syntax conventions and rules still hold true. Being that C++ is the extension and spiritual successor to C, C++ expands by offering, among other things object oriented programming. C++ also simplifies certain tasks through better syntax and libraries.

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++.

Some differences between C++ and C are:

C++	C
function overloading	absent in C89, allowed only for some standard library code in C99.
adds keyword **class**, but keeps **struct** from C, with compatible semantics.	
access controls for class members.	
inheritance and polymorphism	
generic programming through the use of templates.	
extends the C89 standard library with its own standard library.	
different complex number facilities.	only available since C99
bool and **wchar_t** as primative types	as typedefs
comparision operators return **bool**	comparision operators returns **int**.
character constants have type **char**	character constants have type **int**.
additional cast operators (**static_cast**, **dynamic_cast**, **const_cast** and	

reinterpret_cast).	
adds **mutable** keyword to address the imperfect match between physical and logical constness.	
extends the type system with *references*.	
member functions, **constructors** and **destructors** for user-defined types to establish invariants and to manage resources.	
runtime type identification (RTTI), via **typeid** and **dynamic_cast**.	
exception handling.	
has `std::vector` as part of its standard library	uses variable-length arrays instead
treats **sizeof** operator as compile time operation	allows sizeof be a runtime operation.
new and **delete** operators	uses **malloc** and **free** library functions exclusively.

Java

This is a comparison of the Java programming language with the C++ programming language. While C++ and Java share many common traits.

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	protects the programmer
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:

- Java syntax is context-free and can be parsed by a simple LALR parser. Parsing C++ is somewhat more complicated; 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++ 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 methods (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

- Java has a considerably larger standard library than C++. The C++ standard library only provides components that are relatively general purpose, such as strings, containers, and I/O streams. The Java standard library includes

components for networking, graphical user interfaces, XML processing, logging, database access, cryptography, and many other areas. 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 be 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# 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.

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 nitty-gritty 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 (Microsoft Intermediate Language). The CLI code is further reduced to bytecode and executed within the .NET's virtual machine, known as the Common Language Runtime (CLR).

C++/CLI is an extended version of C++ designed to replace Managed C++.

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: High-level and Low-Level:

High-level Languages

High-level programming languages are so named because their syntax is written in a form that can be read and generally interpreted by the average human being. Most of them use words taken from the English language so that they can be written and read without too much difficulty. No programming language is written in what one might call "plain English" though, (although BASIC comes close), so written programs are sometimes referred to as "code", more specifically as "source code".

C++ is a high-level language, as are most of the common ones you may have heard of, BASIC, Perl, Java and others are all high-level languages. A program written in one of these languages is also sometimes referred to as "human-readable code".

Low-level Languages

There are two types of "languages" that fall under the general category of "low-level", though only one is really at all usable by human-beings. The first one, most people have at least heard of, even if they aren't entirely familiar with what it is. It's called binary, and also goes by the name of "machine-code".

It is called machine code because it is really the only thing that the computer actually understands. It consists of strings of 0 and 1, which combine to form meaningful instructions to the computer. It should be 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?

The other "low-level" language is called Assembly language. Assembly language, (also called ASM), is a human-readable translation of the machine language instructions the computer executes. For example, instead of referring to a processor instruction by its binary representation, the programmer will use its mnemonics. These mnemonics are

usually short collections of letters that symbolize the action of the respective instruction. It should be noted that Assembly language is *Processor Specific* (because it's just a set of mnemonics to facilitate machine language programming), which means that a program written in ASM for one computer will not work on one with a different processor architecture, because the processor in the other computer will interpret the instructions differently (most likely as invalid instructions).

Translating Programming Languages

Programmers write programs in order to give a computer instructions, but since the computer is only capable of understanding 1's and 0's, we have to convert our human-readable code into machine code. There are two ways that this is usually done.

The first is by means of an "interpreter". An interpreter is a program that reads a high-level program one line of text at a time and does what it says. In effect, it translates the program line-by-line, alternately reading lines and carrying out commands. Some languages are almost always interpreted, such as BASIC or Perl. Other languages are almost never interpreted, such as C# or C++.

The second method is by means of "compilation". Compilation can be thought of as a two step process. The first step is to "translate" as an interpreter would do, from human-readable to machine-language. The second step is to "package" that machine-code without actually executing any of the instructions, so that the instructions can be executed later. C++ is rarely ever interpreted; instead it is compiled and then "executed" later. As complicated as that may seem, most IDE's (integrated development environment, the place where you write your code), include a "compiler" that will do all of this for you at the click of a button.

High-Level vs. Low-Level

While it is technically possible to create more "efficient" code if you write it in pure assembly-language, the advantages of writing in a high-level language format far outweigh any drawbacks, and those advantages include:

- Advanced program structure: loops, functions, and objects all have limited usability in low-level languages, as their existence is already considered a "high" level feature; that is, each structure element must be further translated into low-level language.
- Portability: high-level programs can run on different kinds of computers with few or no modifications. Low-level programs often use specialized functions available on only certain processors, and have to be rewritten to run on another computer.
- Ease of use: many tasks that would take many lines of code in assembly can be simplified to several function calls from libraries in high-level programming languages. For example, Java, a high-level programming language, is capable of painting a functional window with about five lines of code, while the equivalent assembly language would take at least four times that amount.

For the purposes of this book, we'll only be dealing with one high-level language, and much of your compilation worries will be taken care of by your IDE, but it's still good to have an idea of what's going on "behind-the-scenes".

What is a Program?

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

Types of Programs

There are all kinds of different programs used today, for all types of purposes. What they all have in common is that they were all written with some form of programming language, and they all give the computer instructions of one type or another. Examples of different types of programs, (also called software), include:

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

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

Web Browsers & Email Clients
: A web-browser is a program that allows you to type in an Internet address and then displays that page for you. An email client is a program that allows you to send, receive and compose email messages outside of a web-browser. Often email clients have some capability as a web-browser as well, and some web-browsers have integrated email clients. Well-known web-browsers are Internet Explorer and Firefox, and Email Clients include Microsoft Outlook and Thunderbird.

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

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

Development Software

Development software is software used specifically for programming. It includes software for composing programs in a computer language (sometimes as simple as a text editor like Notepad), for checking to make sure that code is stable and correct (called a debugger), and for compiling that source code into executable programs that can be run later (these are called compilers). Often times, these three separate programs are combined in to one bigger program called an IDE (Integrated Development Environment). There are all kinds of IDEs for every programming language imaginable. A popular C++ IDE for Windows and Linux is the Code::Blocks IDE . The one type of software that you will learn the most about in this book is Development Software.

Types of instructions

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

Input

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

Output

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

Math/Algorithm

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

Testing

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

Repetition

Perform some action repeatedly, usually with some variation.

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

Statements

Most programming languages have the concept of a statement. A statement is a command that the programmer gives to the computer. For example:

cout << "Hi there!";

This command has a verb ("cout") and other details (what to print). In this case, the command "cout" means "show on the screen," not "print on the printer." The programmer either gives 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 could create a file called "hi.txt", put the above command in it, and give the file to the computer.

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

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

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!";
```

Source Code and Source Files

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 on 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".

Source Code

The product of a programmer's efforts is called "source code". It is the actual text that makes up the functions (actions) that the computer must execute. 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.

Just as an aside that will be discussed in more detail later: 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, and the reasons why will be discussed in the section on coding style.

Source File

Source code is nothing more than the text that makes up the commands issued to a computer. For that reason, it is perfectly possible to write source code in a simple text editor like Notepad. When you save that file, it will be saved as filename.txt, right? This is ok if all you want to do is store the code somewhere where you can retrieve it and work on it again. However, if you want to be able to work on it again *and* allow a compiling program to recognize it, you have to designate it as a *Source File* for the language it is written in.

Most operating systems require C++ source files to be designated by a specific extension. The most common extension for a C++ source file is ".cpp" for implementation code, (the code that the programmer wrote to perform a given task), and ".h" for declaration or "header" files. 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 source 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 compiler that the text within the file is C++ source code, and not just random characters.

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 these qualify as source files. 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".

Organizing Code

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 implementation code across many source files. This makes it easier to read, as you can do it bit by bit, and it also reduces compile time. 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 source 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.

Programming Paradigms

A **programming paradigm** is a style or model of programming that affects the way programmers can design, organize and write programs. C++ is designed to allow programmers to use several different programming paradigms which makes it a multiparadigm programming language. A **multiparadigm programming language** gives programmers a choice in designing programs from a number of different programming paradigms.

Procedural Programming

Procedural programming is a programming paradigm based upon the concept of the modularity and scope of program code (i.e., the data viewing range of an executable code

statement). A main procedural program is composed of one or more modules (also called packages or units), either written by the same programmer or provided in a code library by another programmer.

Each module is composed of one or more subprograms (which may consist of procedures, functions, subroutines or methods, depending on the programming language). It is possible for a procedural program to 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

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 methods as adjectives describing that noun, and its members 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

Inheritance describes a relationship between two types, or classes, of objects in which one is said to be a "subtype" or "child" of the other. One object may be based on those of other objects, from which the former object is said to *inherit*. The child inherits features of the parent, allowing for shared functionality. 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.

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.

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.

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 object can inherit the properties of two or more objects.

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.

Generic Programming

Generic programming or **polymorphism** is a programming style that emphasizes techniques that allows 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

Static typing refers to how a computer language handles its variables. Variables are values that the program uses during execution. These values can change; they are variable, hence their name. Static typing usually results in compiled code that executes more quickly. When the compiler knows the exact types that are in use, it can produce machine code that does the right thing easier. In C++, variables need to be defined before they are used so that compilers know what type they are.

Static typing usually finds type errors more reliably at compile time, increasing the reliability of compiled programs. Simply put, it means that "A round peg won't fit in a square hole", so the compiler will report it when a type leads to ambiguity or incompatible usage. However, programmers disagree over how common type 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 and to small bug databases. The value of static typing, then, presumably increases as the strength of the type system is increased.

A statically typed system constrains the use of powerful language constructs more than it constrains less powerful ones. This makes powerful constructs harder to use, and thus places the burden of choosing the "right tool for the problem" on the shoulders of the programmer, who might otherwise be inclined to use the most powerful tool available. Choosing overly powerful tools may cause additional performance, reliability or correctness problems, because there are theoretical limits 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++.

This free-form nature of C++ is used (or abused, depending on your point of view) by some programmers with an unusual sense of humor 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 by hackers, short of using the whole program exactly how it was originally written.

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**. The *compilation* output of a compiler from translating or *compiling* a program is saved to a file called an **object file**. Object files contain the *binary language* (*machine language*) instruction to be used by the computer to do as was instructed in the *source code*. 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 step required to create an executable program.

Linking is the process of connecting or combining object files produced by a compiler with the libraries necessary to make a working executable program. *Linkage* refers to the way in which a program is built out of a number of translation units. A **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. Symbols or identifiers which are needed outside a single translation unit must have external linkage. Of course, the process is more complicated than this; but the basic ideas apply.

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.

ISO C++ (C++98) Keywords

ISO C++ (C++98) keywords:

- `and`
- `and_eq`
- `asm`
- `auto`
- `bitand`
- `bitor`
- `bool`
- `break`
- `case`
- `catch`
- `char`
- `class`
- `compl`
- `const`
- `double`
- `dynamic_cast`
- `else`
- `enum`
- `explicit`
- `export`
- `extern`
- `false`
- `float`
- `for`
- `friend`
- `goto`
- `if`
- `inline`
- `not`
- `not_eq`
- `operator`
- `or`
- `or_eq`
- `private`
- `protected`
- `public`
- `register`
- `reinterpret_cast`
- `return`
- `short`
- `signed`
- `sizeof`
- `this`
- `throw`
- `true`
- `try`
- `typedef`
- `typeid`
- `typename`
- `union`
- `unsigned`
- `using`
- `virtual`
- `void`
- `volatile`
- `wchar_t`

- **const_cast**
- **continue**
- **default**
- **delete**
- **do**
- **int**
- **long**
- **mutable**
- **namespace**
- **new**
- **static**
- **static_cast**
- **struct**
- **switch**
- **template**
- **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.

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

To avoid conflicts on the naming of C++ keywords by vendors, library creators and users in general there are some nonstandard identifiers that are reserved for distinct uses.

Compiler keywords

A limited set of keywords exist to directly control the compiler behavior, these keywords are very powerful and must be used with care they may make a huge difference on the program compile time and running speed.

auto

The `auto` keyword informs the compiler that the variable will go out of scope as the program exits from the current block (a function, a structure or class). All variables declared are by default declared as type `auto`.

```
auto int var;// variable declared using the auto specifier
```

this declaration is equal to...

```
int var;// variable declared without the storage class specifier but
```

```
        // uses auto type specifier by default
```

inline

The idea behind the `inline` keyword is 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. Marking a function as "inline" (possibly implicitly, by defining it inside a class definition) is a (non-binding) request to the compiler to consider inlining the function, i.e., expanding its code at the call site.

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.

Example:

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

extern

The extern keyword tells the compiler that a variable is declared in another source module. The linker then finds this actual declaration and 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 global* `int i` *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.

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

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

#include

The **#include** directive allows a person 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++ requires you to *declare* what will be used before using it. So, files called **headers** generally 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 look like this:

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

The above lines causes the inclusion of the contents of iostream and others.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` 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 Headers

Standard C++ headers

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

Standard C++ headers based on the C Standard Headers.

- `cassert`
- `cctype`
- `cerrno`
- `cfloat`
- `ciso646`
- `climits`
- `clocale`
- `cmath`
- `csetjmp`
- `csignal`
- `cstdarg`
- `cstddef`
- `cstdio`
- `cstdlib`
- `cstring`
- `ctime`
- `cwchar`
- `cwctype`
- `stdiostream.h`[1]
- `stream.h`[2]
- `strstream.h`[3]

Old compilers may include headers named `<X.h>` and `<cX.h>` in addition to or instead of `<X>` and `<cX>` as Standard C++ now requires. `<X>` headers keep everything in the `std::` namespace while `<X.h>` pollutes the global namespace. Many compilers may also have a non-templatized class library for vectors, strings, complex numbers, etc. Some have the SGI STL on which many of the standard headers are based.

[1]Streams based on `FILE*` from `stdio.h`.
[2]Very very old streams library (precursor to `iostream.h`) mostly included only for backward compatibility even on old compilers.
[3]Uses **`char`**`*` whereas `sstream` uses `string`.

#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.

#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 undefines a previously defined value or macro used by the preprocessor.

```
#undef USER_MAX
```

NOTE:
Because preprocessor definitions are substituted before the compiler acts on the source code, any errors that are introduced by `#define` are difficult to trace. For example using value or macro names that are the same as some existing identifier can create subtle errors, since the preprocessor will substitute the identifier names in the source code.

Today, for this reason, `#define` is primarily used to handle compiler and platform differences. E.g, a define might hold a constant which is the appropriate error code for a system call. The use of `#define` should thus be limited unless absolutely necessary; typedef statements, constant variables, enums, templates and inline functions 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 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 \ token.

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

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 it was expanded 3 times (in this case) and thus has an undefined value (which means that the compiler can do anything it wishes).

- ABSOLUTE_VALUE(z++); expanded:

```
( ((z++) < 0 ) ? -(z++) : (z++) );
```

- An example on how to use a macro correctly:

```
#include <iostream>
using namespace std;

#define SLICES 8
#define ADD(x) ( (x) / SLICES )

int main() {
  int a = 0, b = 10, c = 6;

  a = ADD(b + c);
  cout << a;

  return 0;
}
```

-- the result of "a" should be "2" (b + c passed to ADD -> ((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 really right 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

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

into

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

Using ## concatenates what's before the ## with what's after it. For example

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

will make the compiler turn

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

into

```
cout << xy << endl;
```

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

Conditional compilation

- **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 true 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.

#ifdef and #ifndef

The **#ifdef** and **#ifndef** directives are short forms of '**#if** defined(*defined-value*)' and '**#if** !defined(*defined-value*)' respectively. **defined**(*defined-value*) returns true if a preprocessor variable by the name *defined-value* was defined with #define and false otherwise..

#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 "#pragma once".

Compile-time warnings and errors

- **Syntax**:

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

#warning and #error

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

- **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
```

Debugging

If you want to use a debugger (for *debugging*) 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.

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 the process of tracking them down and correcting them is called debugging.

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.

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.

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.

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).

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 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 encouters 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 defintions.

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.

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 )
```

- 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++;
```

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. This set of rules enables the understandings and transparency of the code base, provides for an undocumented structure, easy debugging, and can 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 a broader view as you can get so 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.

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.

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). Please 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.

Functions and Methods

The name given to functions and methods should be descriptive and make it clear what it does. Since usually functions and methods 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 methods can sometimes make guessing correctly what functions and methods 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*

Comments

The C++ standard has this to say about comments:

> Definition:
>
> C++ 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.

> Definition:
>
> C-style comments start with `/*` and end with `*/`

This describes how a comment can be added to the source code, but not where how and when to comment.

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.

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.

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.

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 have a specific comment layout.

Automatic Documentation

Some programming tools can read structured information in comments in order to generate documentation automatically. An example is *doxygen*, which was modeled after JavaDoc.

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.

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. 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
*/
```

if you use this kind of comment try to use it like this...

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

this opens you the option to do this...

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

... 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 `*/`.

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 using both solutions in release code but may use it to do a quick test/debug.
The use of C-style comments should be avoided as they are considered outdated..
Mixing C and C++ style comments may be considered poor practice.

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

if your comment is a "comment box" ie does not contain any line of code you can

use C style to make "beautiful"

```
 /** this is a comment of function XXXX
  * this function do
  *    - tests XXX
  *    - allocated ZZZ
  */
```

try to avoid using C style inside a function because of the non nesting facility of C style (see below)

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 start with `//` and continue until the end of the line.

Example:

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

or

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

or

```
//  CCCC CCCC CCC  CCCC     \ /
//  CC   C CC CC C CCC       *
//  CCCC CCCC CCC  CCCC     / \
```

```
//                          /   \
//  If you wish to use in multiple lines
```

Don't end a comment line with the backslash \ character; 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;

Whitespace and Indentation

Definition:

> Spaces, tabs and newlines (line breaks) are called *whitespace*. Spaces or tabs are required to separate adjacent words and numbers; they are ignored everywhere else except within quotes. Newlines are ignored in all regular statements other than quote, preprocessor directives, and C++-style comments.

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

```
// This program just displays a string and exits
#include <iostream>
int main(){std::cout<<"Hello World!";std::cout<<std::endl;return 0;}
```

However, the same program could be made much more readably with proper indentation:

```
// This program just displays a string and exits
#include <iostream>

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

To make the program more readable, suitable whitespace must be used.

Indent size

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 tabs differently and expand them differently. Never combine tabs and spaces, use either or one of them. The K&R standard recommends an indentation size of 2 spaces.

The conventions followed when using whitespace to improve the readability of code constitute an Indent style.

Every block of code and every definition should be indented with the indentation size. This usually means everything within `{` and `}`. However, the same thing goes for one-line code blocks.

Examples:

```
// Using an indentation size of 2
```

```
if (a > 5) {
  b=a;
  a++;
}
// Using an indentation size of 4
if (b > 6)
    c--;
```

Placement of parentheses

Different Coding styles 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 (Java style). 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 and Java style
}
if (a > 5)
{
  // This is ANSI C++ style
}
if (a > 5)
  {
    // This is GNU style
  }
```

Code Blocks

A code block (or in C++-speak, a *compound statement*) is one or more statements or commands that are contained between a pair of curly braces `{ }`. 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.

File Organization

Return Codes

There are 2 kinds of behaviors :

Positive Means Success

This is the "logical" way to think, and as such the one used by almost all beginners. The derivative form is positive means success, negative or 0 means failure.

The major problem of this construct is that it creates more imbrication, for example:

```
if (my_function1 ())
  {
    // block of instruction 1
    if (my_function2 ())
      {
        // block of instruction 2
        if (my_function3 ())
          {
             // block of instruction 3
          }
        else
          {
             //error handler
             return 3;
          }
      }
   else
     {
         //error handler
         return 2;
     }
  }
else
  {
    //error handler
    return 1;
  }
```

As you can see the else block (usually error handling) of my_function1 can be really far from the test; 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 is compensated by features of editors like folding.

The second problem of this construct is that it tends to imbrication tests (my_function2 is one level more indented, my_function3 is 3 levels...) which causes legibility problems.

0 means success

Meaning that if a function return 0 it's successful or else an error code is returned, the advantage of this kind of behavior is that it allow less imbrication of if else constructs, for example the previous code becomes:

```
if ( my_function1() )
 {
    //error handler

 return 1;
 }
// block of instruction 1
if ( my_function2() )
  {
     //error handler
```

```
  return 2;
  }
// block of instruction 2
if ( my_function3() )
  {
     //error handler

  return 3;
  }
//  // block of instruction 3 + i
//  if ( ... )
//    }
//    ...
//    }
```

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. **HOWEVER** this rule is really a good one. In the previous paragraph you saw that the "0 means success" rule prevented imbrication code, so avoid breaking the 80 columns rule.

In fact this is not an **absolute** rule, but this rules means **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**.

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

Basics: Learn the fundamentals

Namespace

In many programming languages, a namespace is a context for identifiers. C++ can handle multiple namespaces within the language.

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 name space’.

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`.

`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 `food::orange` if it 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 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 qualifiing, 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.

Although you can nest an anonymous namespace in an ordinary namespace, and vice versa, you can also nest two anonymous namespaces.

```
namespace {
  namespace {
    // ok
  }
}
```

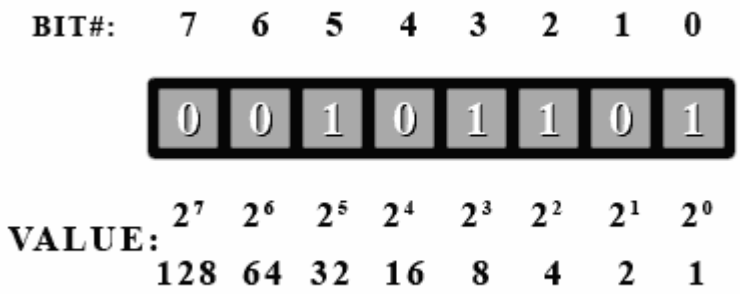
NOTE:
If you enable the use of a `namespace` in the code, all the code will use it (you can't define sections that will and exclude others), you can however use nested `namespace` declarations to restrict its scope.

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 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)

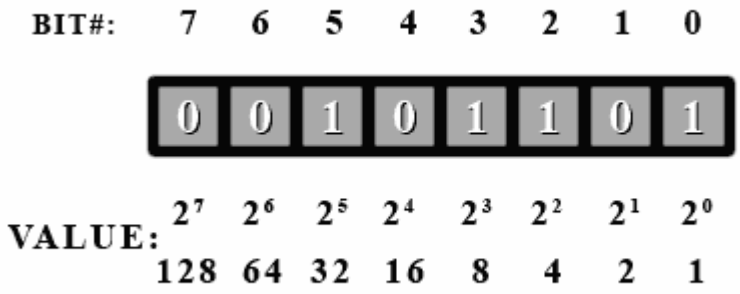
All this is elementary. The key point to recognize is that as we move from right to left in the number, the significance of the digits increases by a multiple of 10. This should be obvious when we look at the following equation:

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

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.

$00_2 = 0_{10}$

$01_2 = 1_{10}$

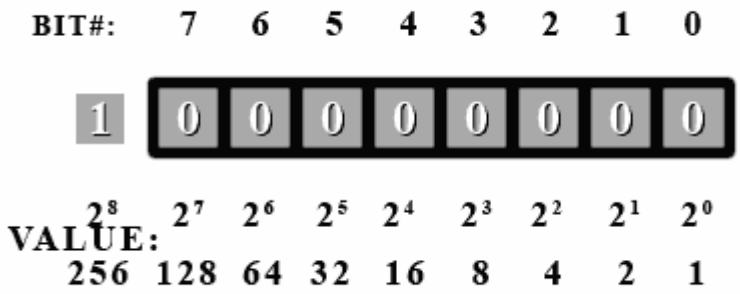
$10_2 = 2_{10}$

$11_2 = 3_{10}$

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.

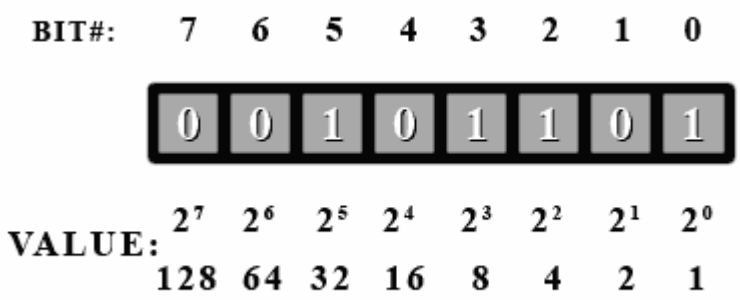
If you were to run this program, you would notice something strange. After 127, we got -128! Negative numbers! Where did these come from? Well, it just so happens that the compiler needs to be told if we're using numbers that can be negative or number that can only be positive. These are called signed 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

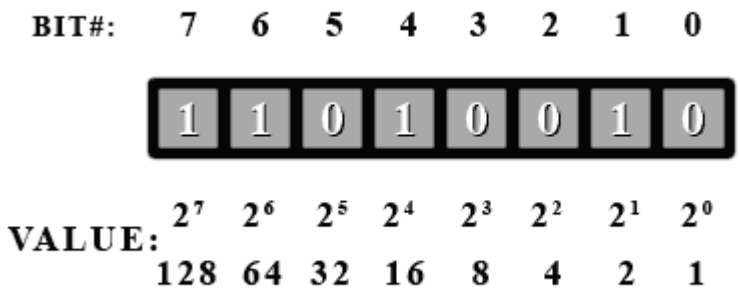
Two's complement is a method on how to store negative numbers. The reason that the two's complement method of storing negative numbers was chosen is because this allows the CPU to use the same add and subtract instructions on both signed and unsigned numbers.

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).

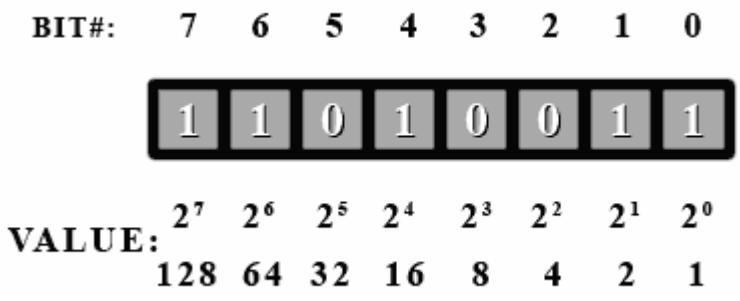
Let's try to convert our number 45.



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.

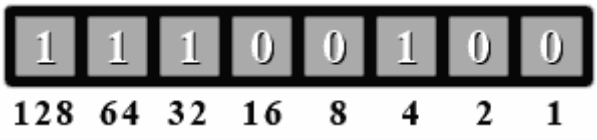
If and only if you decide to treat it as a signed number, then look at the msb (most significant bit [bit#7]). **If it's a 1, then it's a negative number.** If it's a 0, then it's

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?

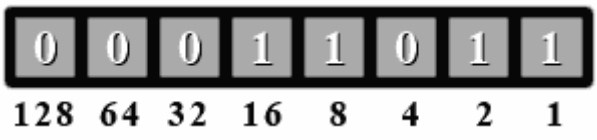


Since this is an unsigned number, no special handling is needed. Just add up all the values where there's a 1 bit. 128+64+32+4=228. So this binary number is 228 in decimal.

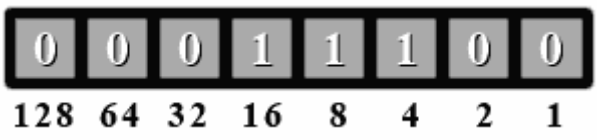
Now treat the number above as a signed byte. What is its value in decimal?

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...



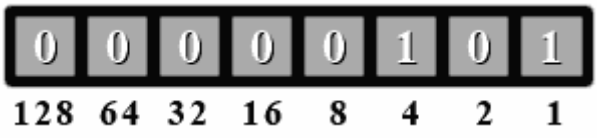
And add 1. This is a little trickier since a carry propagates to the third bit. For bit#0, we do 1+1 = 10 in binary. So we have a 0 in bit#0. Now we have to add the carry to the second bit (bit#1). 1+1=10. bit#1 is 0 and again we carry a 1 over to the 3rd bit (bit#2). 0+1 = 1 and we're done the conversion.



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 4 bytes or 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, max unsigned value: 255)

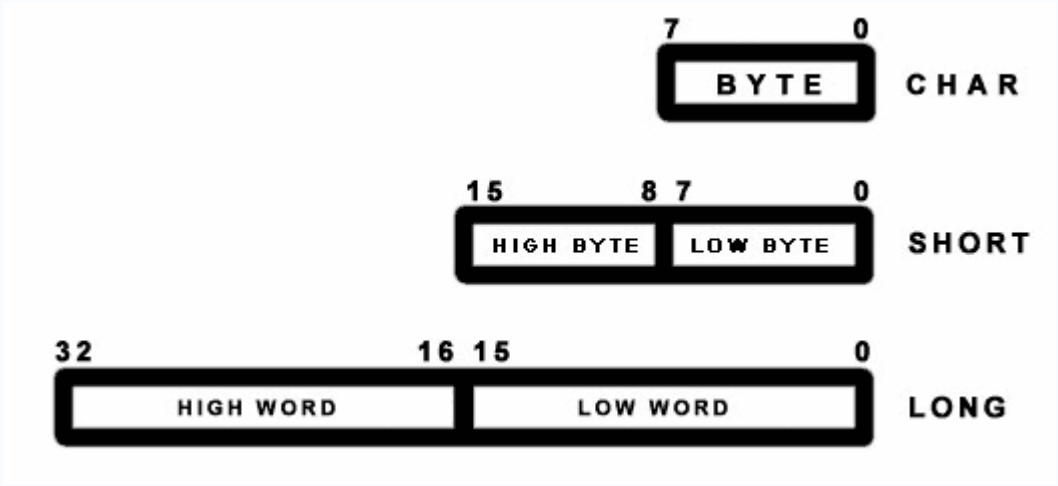
2. short int (2 bytes, max unsigned value: 65535)

3. long int (4 bytes, max unsigned value: 4294967295)

4. float (4 bytes, floating point)

5. double (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, an int defaults to a 'long int' (32 bits).



A word is the same as a short (16 bits). The term 'word' is used more commonly in hardware & assembler technical manuals.

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.

Let's look at a 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 CPU's do work this way. The Amiga and old Macintoshes use the M68000 and they indeed do use this byte ordering.

However, on PC's (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 PC's. 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 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-endien 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-endien), 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 tranformed 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.

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,"/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.

standard types

C++ has five basic primitive types called **standard types**, specified by particular keywords, that store a single value. For information on the size of the different types, see C++ Data Types. 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, version 1.
#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, version 2
#include <iostream>
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. 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`.

Enumerative 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 a 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; month <= 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**.

References

References are a way of assigning a "handle" to a variable.

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 C++ 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 variable references have is to pass function arguments using references.

A Program Using Variables

Below is a C++ 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;
}
```

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.

Type casting

Type checking is the process of verifying and enforcing the constraints of types. This may occur either at compile-time (a static check) or run-time (a dynamic check). Static type checking is a primary task of the semantic analysis carried out by a compiler. If type rules are enforced strongly (that is, generally allowing only those automatic type conversions which do not lose information), the process is called *strongly typed*, if not, *weakly typed*. C++ is strongly typed, but allows programmers to interpret or use a variable or expression of one type as if it were another type, this is called **type casting**.

Automatic type casting

Whenever the compiler expects a data of a particular type, but the data is given as a different type, it will try to automatically type cast.

Examples:

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

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 cast: promotion and demotion.

Promotion

Promotion occurs whenever a variable or expression of a *smaller* type gets cast to a *larger* type.

Examples:

```
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 cast to a *smaller* type.

Examples:

```
int a = 7.5;   // float gets downcast to int;
int b = 7.0;   // float gets downcast to int;
char c = b;    // int gets downcast 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 compiler will generate a warning if demotion occurs. Should the loss of information be intended, the programmer should do explicit type casting to suppress the warning.

Explicit type casting

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

The basic form of type cast

The basic explicit form of type casting 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:

Example:

```
int a = static_cast<int>(7.5);
```

can be used to suppress the warning shown above.

Advanced type casts

const_cast

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

static_cast

static_cast<T>(expression)
The static_cast<>() is used to cast between the integer types.
'eg' 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. 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) or (type)expression. The format of (type)expression is more common in C. It has the basic form:

```
int i = 10;
long l;

l = (long)i;
// or
l = long(i);
```

(Note: Technically the `long(i)` form is not a typecast; and it instead invokes the constructor of **long**. For primitive types like **long**, the *constructor* is effectively the same as type casting; but this may not be the case for class types which have user-defined *constructors*.)

Expression can be any C++ expression. It can work in places of both *static* and *dynamic cast* (not sure about other types of cast though)

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 - converts between two unrelated types

```
c = 'a';
int b = reinterpret_cast<int>(c);
```

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

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

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

Operators

Programming languages have a set of operators that perform arithmetical operations, and others such as Boolean operations on truth values, and string operators manipulating strings of text. 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.

Conventionally, the computing usage of *operator* also goes beyond the mathematical usage (for functions). The C programming language syntax for pointers, uses the *operators* `*` and `&`. `sizeof` is sometimes considered an operator, and in C++, `new` and `delete` are also operators, also in C++, you can define your own uses for operators. When an operator is alphanumeric rather than a punctuation character, it is sometimes called a *named operator*.

So operators are special symbols that are used to represent for example simple computations like addition and multiplication. 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 `+`.

The following are all legal C++ 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;
cout << "Number of minutes since midnight: ";
cout << hour*60 + minute << endl;
cout << "Fraction of the hour that has passed: ";
cout << minute/60 << 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:

```
cout << "Percentage of the hour that has passed: ";
cout << minute*100/60 << 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.

Assignment

The use of the "=" assignment operator.

Arithmetic operators

The line:

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

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

The use of the "+=", "-=", "*=", "/=", "%=", "<<=", ">>=", "^=", "|=" and "&=" compound assignment...

Conditional operators

Conditional operators (also known as **logic** and **boolean** operators) allow the expression of mathematical conditions. These 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...). All conditional operators return a boolean value, true or false depending if the condition is true or the condition is false.

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, allow 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 no if ((x>5) && !(x == 10))
{
  //...code...
}
```

When all conditions must be true. If x must be between 10 and 20:

```
if (x > 10 && x < 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)
{
  //...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))
{
  //...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 thing 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.

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.

Operators for characters

Interestingly, the same mathematical operations that work on integers also work on characters. For example,

```
char letter;
letter = 'a' + 1;
cout << letter << endl;
```

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';
cout << number << 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.

Composition (Operator Precedence)

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:

```
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:

```
cout << hour*60 + minute << 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.

WARNING: 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. So 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 in C++ as it was in C; operator overloading and reference types complicate the picture.)

Special Operators

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; // create an array of 4 chars
```

and so on...

to initialize as you declare them you can use:

```
int vector={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;
```

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 = { 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;
```

To access/modify a value in this array we need two subscripts:

```
 char ch;
 ch = two_d;
```

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) Ease of use/port: Integrated into most modern languages

Disadvantages include:

- Constant size
- Constant data-type
- Large free sequential block to accommodate large arrays

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, can be dynamically resized, use 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 are is similar to the *array operator*, `array[]` is equal to `array*`.

t x = 10, it would be written:

Pointers ("*") and References ("&")

The easiest way to explain pointers and references is to jump right into an example. Let's first take a look at some Algebra: `x = 1`

In Algebra, when you use a variable, it is essentially a letter or designation that you use to store some number. In programming, the variable in the equation above must be on the left side. You've probably noticed by now that the compiler won't let you do something like this: `1 = x;`

And if you didn't know this...now you know, and knowing is half the battle. The reason why you receive a compile-time error like "lvalue required in..." is because the left hand side of the equation, traditionally referred to as the lvalue, must be an address in memory.

Think about it for a second. If you wanted to store some data somewhere, you first need to know where you're going to store it before the action can take place. The lvalue is the address of the place in memory where you're going to store the information and/or data of the right hand side of the equation, better known as the rvalue.

In C++, you will most likely at one point or another, deal with memory management. To manipulate addresses, C++ has two mechanisms: pointers and references.

What are pointers and references?

Pointers are essentially variables that hold memory addresses as their values. This allows us to use a pointer when we don't know at compile-time which actual object we will want to operate on; instead of using a named variable directly, we use a pointer and set it to "point" to the right object when the program runs.

References can also be used for some of the same things as pointers. A reference is an "alias", i.e., another name for some object. In spite of having some uses in common with pointers, references are best considered as completely separate things. There are some things that can be done with references but not with pointers (e.g., implementing a copy constructor), and some things that can be done with pointers but not references (e.g., a reference cannot be NULL, a reference cannot be "re-seated").

You learned before about the various different data types such as: int, double, and char. Pointers hold the addresses in memory of where you find the data of the various data types that you have declared and assigned. The two mechanisms, pointers and references, have different syntax, semantics, and different traditional uses.

Declaring pointers and references

When declaring a pointer to an object or data type, you basically follow the same rules of declaring variables and data types that you have been using, only now, to declare a pointer of `SOMETYPE`, you tack on an asterisk * between the data type and its variable.

```
SOMETYPE* sometype;
int* x;
```

To declare a reference, you do the exact same thing you did to declare a pointer, only this time, rather than using an asterisk `*`, use instead an ampersand `&`.

```
SOMETYPE& sometype;
```

This should be read as "declare sometype to be a variable of type reference-to-SOMETYPE".

```
int& x;
```

("x is a reference to an int")

As you probably have already learned, spacing between tokens in C++ does not matter (after preprocessing), so the following pointer declarations are identical:

```
SOMETYPE*  sometype;
SOMETYPE * sometype;
SOMETYPE  *sometype;
```

The following reference declarations are identical as well:

```
SOMETYPE&  sometype;
SOMETYPE & sometype;
SOMETYPE  &sometype;
```

The "address of" operator

Although declaring pointers and references look similar, assigning them is a whole different story. In C++, there is another operator that you'll get to know intimately, the "address of" operator, which is (also) 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.

```
SOMETYPE* x = &sometype; // 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.

```
SOMETYPE MyFunc(SOMETYPE *x)
{
  cout << *x << endl;
}

int main()
{
  SOMETYPE i;
  MyFunc(&i);
  return 0;
}
```

Assigning pointers and references

As you saw in the syntax of using the "address of" operator, a pointer is assigned to the return value of the "address of" operator. Because the return value of an "address of" operator is a pointer, everything works out and your code should compile. To assign a pointer, it must be given an address in memory as the rvalue, else, the compiler will give you an error.

```
int x;
int* px = &x;
```

The above piece of code shows a variable x of type int being declared, and then a pointer px being declared and assigned to the address in memory of x. The pointer px essentially "points" to x by storing its address in memory. Keep in mind that when declaring a pointer, the pointer needs to be of the same type pointer as the variable or constant from which you take the address.

Now here is where you begin to see the differences between pointers and references. To assign a pointer to an address in memory, you had to have used the "address of" operator to return the address in memory of the variable as a pointer. A reference however, does not need to use the "address of" operator to be assigned to an address in memory. To assign an address in memory of a variable to a reference, you just need to use the variable as the rvalue.

```
int x;
int& rx = x;
```

The above piece of code shows a variable x of type int being declared, and then a reference rx being declared and assigned to "refer to" x. Notice how the address of x is stored in rx, or "referred to" by rx without the use of any operators, just the variable. You must also follow the same rule as pointers, wherein you must declare the same type reference as the variable or constant to which you refer.

Hypothetically, if you wanted to see what output a pointer would be...

```
#include <iostream>

int main()
{
  int someNumber = 12345;
  int* ptrSomeNumber = &someNumber;

  std::cout << "someNumber = " << someNumber << std::endl;
  std::cout << "ptrSomeNumber = " << ptrSomeNumber << std::endl;

return 0;
}
```

If you compiled and ran the above code, you would have the variable someNumber output 12345 while ptrSomeNumber would output some hexadecimal number (addresses

in memory are represented in hex). Now, if you wanted to **cout** the value pointed to by ptrSomeNumber, you would use this code:

```
#include <iostream>

int main()
{
  int someNumber = 12345;
  int* ptrSomeNumber = &someNumber;

  std::cout << "someNumber = " << someNumber << std::endl;
  std::cout << "ptrSomeNumber points to " << *ptrSomeNumber <<
std::endl;

return 0;
}
```

So basically, when you want to use, modify, or manipulate the value pointed to by pointer x, you denote the value/variable with *x.

Here is a quick list of things you can do with pointers and references:

- You can assign pointers to "point to" addresses in memory
- You can assign references to "refer to" variables or constants
- You can copy the values of pointers to other pointers
- You can modify the values stored in the memory pointed to or referred to by pointers and/or references, respectively
- You can also increment or decrement the addresses stored in pointers
- You can pass pointers and/or references to functions (Further information on "Passing by reference" can be found HERE)

For a more in depth view see: Casting (Type)

The Null pointer

Remember how you can assign a character or string to be null? If you don't remember, check out HERE. The null character in a string denotes the end of a string, however, if a pointer were to be assigned to the null pointer, it points to nothing. The null pointer is often denoted by 0 or null. The null pointer is often used in conditions and/or in logical operations.

```
#include <iostream>

int main()
{
  int x = 12345;
  int* px = &x;

  while (px) {
    std::cout << "Pointer px points to something\n";
```

```
    px = 0;
  }

  std::cout << "Pointer px points to null, nothing, nada!\n";

return 0;
}
```

If pointer px is NOT null, then it is pointing to something, however, if the pointer is null, then it is pointing to nothing. The null pointer becomes very useful when you must test the state of a pointer, whether it has a value or not.

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 use:

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

new and delete

Dynamic Memory Allocation

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.

Several sections ago, we learned about assigning pointers using the "address of" operator because it returned 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. In C++ 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.

```
int n = 10;
SOMETYPE *parray, *pS;
int *pint;
parray = new SOMETYPE[n];
pS = new SOMETYPE;
pint = new int;
```

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. 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.

While 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`.

Operator Chaining

The line:

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

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

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;
```

Since a single expression statement can contain any complicated mix of operators and values, the program can be shrunk further to:

```
// This program also adds two numbers and is smaller than the other
two!
#include <iostream>

int main()
{
    std::cout << "The sum of 123 and 456 is " << 123 + 456 << "\n";
    return 0;
}
```

but this does not illustrate the use of variables. Also variables are useful when we want to get the value from the user.

Operator Overloading

NOTE:
Operators can be redefined (overloaded), this concept will be extended after the introduction to some other Object Oriented programming concepts and to classes.

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.

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 *for a type*	**`typeid`** `(std::iostream)`	
`static_cast``<>()`	new C++ style cast operator *for compile-time type conversion*		
`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 *for dynamically allocated arrays*	**`new int`** `[100]`	
`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		
`+ -`	addition and subtraction operators	`end - start + 1`	Left to right
	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		
`&&`		`arr != 0 && arr->len != 0`	Left to right

	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

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 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).

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 some sort of 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 in 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; statement3;
if (condition) statement; else statement2; statement3;
```

- **Semantic**:

First, the condition is evaluated:

- if *condition* is true, *statement* is executed before continuing with *statement3*.
- if *condition* is false and there is no else clause executing continues with *statement3*.

- if *condition* is false and there is a else clause *statement2* is executed before continuing with *statement3*.

Remark: *statement* and *statement2* can be a block of code { ... } with several instructions.

Sometimes only one path is needed depending on a condition. This example prints the message if 'user_age' is less then 18.

```
 if (user_age < 18)
 {
    std::cout << "People under the age of 18 are not allowed." <<
std::endl;
 }
```

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 its in.

switch (Multiple branching)

The switch statement branches based on specific integer values.

```
switch (integer expression) {
    case constant integer expr:
         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 = coin.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 a **goto** 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.)

Loops (iterations)

A loop (also referred to as an iteration) 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

Again, most programming languages have constructions for repeating a loop until some condition changes. Note that some variations place the test at the start of the loop, while others 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 obeyed at least once.

while (Preconditional repetition)

- **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 curly braces are 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 repetition)

- **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 do - while loop construct moves the test that continues the loop to the end of the code block therefore the code block is executed at least once.

- **Example 1**:

```
#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 (Counter-controlled repetition)

Most programming languages have constructions for repeating a loop a certain number of times.

- **Syntax**:

```
for (initialization ; condition ; incrementing expr)
```

```
statement(s)
```

Remark: *statements(s)* can be a block of code { ... } with several instructions. Also, each step of the loop (initialization, condition, and increment) can have more than one command, separated by a comma.

The for loop in C++ is a very general construct, which can run unbounded loops **#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 **#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.

- **Example:**

```
// calls doSomethingWith() for 0,1,2,..9
for (int i = 0; i != 10; ++i)
{
  doSomethingWith(i);
}
```

Other Control Flow Constructs

goto

The constructs mentioned above suffice to express the majority of algorithms in C++. However, in some cases the programmer needs even more control. This control is afforded by the **goto** statement. It is of the form:

```
goto label;
```

where *label* the name of a labeled statement elsewhere in the function. A labeled statement looks like this:

```
label:
     statement(s)
```

(Note the colon that follows the label name).

Here is example of the use of goto to break out of two nested loops 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 */
```

Using gotos is strongly discouraged because it tends to obfuscate the intent of a program. For instance, the following 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*.

It is often said that using gotos is a bad idea... in **MOST** case this is true. Gotos tend to create illegible code, that is then difficulty maintainable. **However** there is a case where gotos are really useful without creating illegible code : the "error section" case. A function that falls into this case is usually built as follow:

```
int* my_allocated_1;
char* my_allocated_2;
char* my_allocated_3;
my_allocated_1 = malloc (500 * sizeof(int));
if (NULL == my_allocated_1)
   {
      fprintf (stderr, "error in alloacted_1");
      goto error;
   }
my_allocated_2 = malloc (1000 * sizeof(char));
if (NULL == my_allocated_2)
   {
      fprintf (stderr, "error in alloacted_2");
      goto error;
   }
my_allocated_3 = malloc (1000 * sizeof(char));
if (NULL == my_allocated_2)
   {
      fprintf (stderr, "error in alloacted_3");
      goto error;
   }
return 0;
```

```
    error:
         if (my_allocated_1) free (my_allocated_1);
         if (my_allocated_2) free (my_allocated_2);
         return 1;
```

this construct is simpler than the usual and less error prone because you don't have to take care of where the error occurs, you then can keep the code simpler and cleaner.

NOTE:

- this code use GNU indentation style
- this code use reverse error code : 0 means success, any other means failure. This kind of **return** value help keeping the code smaller and cleaner

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 **method**.

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 any arguments that are to be passed to it. Arguments like variables must be named and declare what type of value it takes. Arguments 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.

Example:

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

defines a function named `main` that returns an integer value **int** and passes no values. The contents of the function are called the *body* of the function. The word **`int`** is shown in bold because it is a **keyword**. C++ keywords have some special meaning and are also **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 Function

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 and may *call* other functions which may call yet other functions.

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.

NOTE:
The ISO C++ Standard (ISO/IEC 14882:1998) specifically requires main to return int. But the ISO C Standard (ISO/IEC 9899:1999) actually does not.

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 passing two values as argument to `main`. The first argument 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 always at least one argument passed to a program; the name of the program itself is the first argument, `argv[0]`. Other arguments may get passed from the system. The following program should help you understand this:

```
#include <stdio.h>

int main(int argc, char **argv){
  printf("Number of arguments: %d\n", argc);
  for(int i = 0; i < argc; i++){
    printf("  Argument %d = '%s'\n", i, argv[i]);
  }
}
```

If the program above is compiled into the executable `arguments` and executed from the command line like this:

```
$ ./arguments I love chocolate cake
```

It will output the following:

```
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 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. 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.

Arguments

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

```
void printTime (int hour, int minute) {
 cout << hour;
 cout << ":";
 cout << minute;
}
```

It might be tempting to write (int hour, minute), but that format is only legal for variable declarations, not for arguments declarations.

Another common source of confusion is that you do not have to declare the types of arguments when you call a function. The following is wrong!

```
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 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 actually pointers to memory space. In this code:

```
int my_array;
```

`my_array` is actually a pointer to a memory space 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`. When you write `my_array`, you're actually saying "give me the third integer in the array `my_array`". Therefore, `my_array` is a pointer or an array, but `my_array` 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. Here's a quick example:

```
#include <stdio.h>

void printInt(int printable){
  printf("The int you passed in has value %d\n", printable);
}
int main(){
  int my_array;

  // 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` 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. Let's try it out:

```
#include <stdio.h>

void printIntArr(int *array_arg, int array_len){
  printf("The length of the array is %d\n", array_len);
  for(int i = 0; i < array_len; i++)
    printf("Array[%d] = %d\n", i, array_arg[i]);
}

int main(){
  int my_array;

  // 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, 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)
```

This will output the following:

```
The length of the array is 5
Array[0] = 0
Array = 2
Array = 4
Array = 6
Array = 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. Because that's what arrays are in C++. `my_array` is a pointer to an array. `array_arg` becomes a pointer to that same memory space. If

you change `array_arg` within the function, `my_array` doesn't change (IE: 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 both `array_arg` and `my_array`. So be very careful changing array elements in a function. In fact, just don't do it.

Passing by Reference

The same concept of references is used when passing variables. For 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.

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 )
{
  std::cout << s << std::endl;
}

void bar( std::string s )
{
  std::cout << s << std::endl;
}

int main()
{
```

```
   std::string 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 instance if text contained the text of an entire book.

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 this 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.

Passing by Value

Constant Arguments

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

Returning Values (Results)

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
```

Woah, my a-paul-igies, I didn't mean to jump right into it, but I'm pretty sure that if you're understanding pointers, the declaration of a function that returns a pointer or a reference should seem relatively logical. The above piece of code shows how to declare a function that will return a reference or a pointer.

```
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 that you really do not care about. 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 proper code
  int *p = new int;
  ...
  ...
  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.)

```
const SOMETYPE *MyFunc(int *p)
{
  ...
  ...
  return p;
}
```

in this case the value that is returned may not be modified.

So, if we are using a method with a **const** return value we must assign the value to a **const** local variable.

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 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!

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 (Recursive Functions)

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. A terminating condition is used to prevent infinite recursion.

Example

```
 double power(double x, int n)
 {
  if(n<0)
  {
     cout << 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.

Pointers to functions

To declare a pointer to a function, 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;
(*ptof)(5); // calls func
ptof(5); // same thing.
```

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 arguments 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.

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 arguments 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 argument. 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 argument 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.

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`).

Why should you Use Structs, Not Classes?

Basically, C++ uses structs to comply with C's heritage (the code and the programmers). Structs are simpler to the programmer and to 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 ()
{
  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.

Unions

Unions are the same as a `struct` 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).

What is the point of this? Well, by using unions it provides multiple ways of viewing the same memory location, and thus it is a safer and more efficient way to pass addresses, insead of casting it to another variable address

(Actually the main point of unions is more efficient use of memory; most uses of type punning in unions are not valid in portable C++ code. There's little or no connection between unions and an "efficient way to pass addresses.".)

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 portions { char lo; char hi; };
};
```

Using this union we can modify the low-order or high-order bytes of theItem without disturbing any other bytes. Take this simple usage for example:

```
item tItem;
tItem.theItem = 0xBEAD;
tItem.portions.lo = 0xEF; // The item now equals 0xBEEF
```

Classes

A *class* is used to make a new type defined by the programmer called a *user defined type*. A program can contain any number of classes. An instance of the class is called an *object*. Unlike the built-in types though, an object has some values and functions associated with it. As with other types, object types are case-sensitive.

A class not only can contains data, but also functions/methods that can (and should) be used to interact with data contained within the class. Classes 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. Classes provide *encapsulation* defined in the Object Oriented Programming (OOP) paradigm.

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 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.

Declaration

A class is *defined* by:

```
class myClass {
/* private, public, and protected
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)

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.

Public, Protected and Private Labels

C++ supports three labels within classes to define the permissions for the members in that section of the *class*. The labels can be in any order. These labels can be used multiple times in a class declaration for cases where it's logical to have multiple groups of these types.

We have already mentioned that a class can have methods "inside" it, we will see later more about them. Those methods can access and modify all the data and method that are inside the class. Therefore, permission labels are to restrict access to methods 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 method 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 methods 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 members may be accessed freely. In the code, wherever a declared object is in scope is where an object can access any member within the *public* section of the class definition. Including data members in this section can cause unforeseen disasters in the future when others use your code.

NOTE:
In other words, avoid, like the plague, declaring public data members.

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

Protected label has an special meaning related with inheritance, in the section inheritance we will se more about it. More than it, it is just like private label: only member methods can access members declared in a protected scope.

Member Functions (Methods)

Within classes there are data members and functions. To protect the data members, the programmer can define functions to perform the operations on those data members. Methods 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 .h 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;
}
```

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.

const methods

This type of method cannot modify the member variables of a class. It's a hint both to the programmer and the compiler that a given method 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` method cannot call a non-const method (and the compiler will complain if you try). The `const` method cannot change member variables and a non-const method can change member variables. Since we assume non-const methods do change member variables, `const` methods are assumed to never change member variables and can't call functions that do change member variables.

The following code example explains what `const` can do depending on where it is placed.

```
class Foo
{
public:
   /*
    * Modifies m_widget and the user
    * may modify the returned widget.
    */
   Widget *widget();

   /*
    * Does not modify m_widget but the
    * user may modify the returned widget.
    */
   Widget *widget() const;

   /*
    * Modifies m_widget, but the user
    * may not modify the returned widget.
    */
   const Widget *cWidget();

   /*
    * Does not modify m_widget and the user
    * may not modify the returned widget.
    */
   const Widget *cWidget() const;

private:
   Widget *m_widget;
};
```

Accessors and Modifiers (Setter/Getter)

What is an accessor?

> An accessor is a class method that does not modify the state of an object. The accessor functions should be declared as const operations.

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’.

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{
  private int speed;
  public int getSpeed(){
     return speed;
  }
  public void setSpeed(int speed){
     this.speed = speed;
  }
}
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:

```
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 method getCarSpeed() is called. This is known as Lazy initialization.

Overloading

Methods can be overloaded. This means that multiple methods can exist with the same name, but must have different signatures. A method's signature is comprised of the method's name and the type and order of the method's parameters.

Constructor

A constructor is a special member function used to create new objects. The compiler calls the constructor after the new object has been allocated in memory, and converts that "raw" memory into a proper, typed object. Additionally, constructors are responsible for defining data members upon object instantiation (when an object is declared), if the programmer chooses.

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 especialy 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.

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 are a way of initializing data members other than in the body of the constructor. Constructors for the members are included between the argument list and the body of the function (separated from the argument list by a colon).

Example:

```
MyClass::MyClass(int mySimpleMember, MyComplexClass myComplexMember) :
_myComplexMember(myComplexMember)
{
 _mySimpleMember=mySimpleMember;
}
```

This is probably more efficient than initializing the complex data member inside the body of the constructor because in that case the variable is initialized with its default constructor (the one with no arguments). It is also the only way to initialize a class that has a member with no default 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 compiletime. 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 is also possible for normal functions but very seldom used there.

Destructor

Destructors are crucial in avoiding resource leaks 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. 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.

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.

Static Members

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 method 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);
}
```

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.

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 noncopyable 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 `C` uses dynamically allocated memory, you *must* define the destructor `~C ()`.

Alternatively, when using the Boost library, the `boost:noncopyable` utility class can be used:

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

Inheritance

Inheritance describes a relationship between two or more types or classes of objects in which one is said to be a "subtype" or "child" object inheriting from "base" or "parent"

objects. *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.

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:
         int area;
  public:
         int color;

         int getArea(){
             return this->area
         }

         void setArea(int area){
              this->area=area
         }
}

class Circle: public Form {
  public:
         int getRatio() {
             int a;
             a= getArea();
             return sqrt(a/2*3.14);
         }

         void setRatio(int diameter) {
             setArea(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 private.	The member is public.	The member is protected.
private inheritance	The member is private.	The member is private.	The member is private.
protected inheritance	The member is private.	The member is protected.	The member is protected.

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 no 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.

Subsumption

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:

```
enum animalType {
     Herbivore;
     Carnivore;
     Omnivore; //eats humans
}

class animal {
      public:
             bool isAlive;
             animalType Type;
             int numberOfChildren;
}

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

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->numberOfTits=2; //Correct

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

    cat* c= (cat*)a3; //Downcast, correct
    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. Downcasts should be avoided in other circumstances.

Polymorphism

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 form the base class, but to have a different implementation for it? That is when we are talking about polymorphism, a fundamental concept in opp programming.

The concept of polymorphism is wider. Polymorphism exists every time we use two methods that have the same signature, but differ in the implementation.

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 the instruction b.c() should 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` methods 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.

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 methods

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 be required to provide an implementation on their own. This is the case for "pure" virtuals.

To indicate a "pure" `virtual` method 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:
   virtual void paint()
   {
       // do some stuff to draw a button
   }
};
```

Because `paint()` is a pure `virtual` method in the Widget `class` we are required to provide an implementation in all 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 method for the "real" widget type -- not `Widget::paint()` (which is a "pure" `virtual` and will cause the program to crash if called). By using a `virtual` method we insure that the method implementation for our subclass -- `Button::paint()` in this case -- will be called.

NOTE:
While it is not required to use the virtual keyword in our subclass implementations (since if the base class implementation is virtual all subclass implementations will be virtual) it is still good style to do so.

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 Bar (); }
    // Virtual copy constructor, inline implementation
    Bar* clone() const { return new Bar (*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.

Covariant return types

Covariant return types is the ability for a virtual function in a derived class to return 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
{
public:
 virtual derived create() const;
};
```

This allows casting to be avoided.

Abstract Classes

An abstract class is, conceptually, a class that cannot be instanced. There is no way to express this concept directly in C++ (other languages, such as Java provide such constructions). In C++, an abstract class is a class that has one or more pure virtual (abstract) functions in the class. A pure virtual function is a virtual function that has no implementation in the class (it could be defined in classes that inherit from the current class). This is indicated in the declaration with an assignation to zero.

Example

```
   class AbstractClass {
   public:
     virtual void AbstractMemberFunction() = 0; //pure virtual function
     virtual void NonAbstractMemberFunction1(); //virtual function
     void NonAbstractMemberFunction2();
   };
```

In general an abstract class is used to define an implementation 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. If we wish to create a concrete class (a class that can be instanced) 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++.

Example

```
class VehicleObject {
int m_topSpeed;
public:
    VechileObject(int topSpeed)
    : m_topSpeed(topSpeed)
    {}
    int TopSpeed() {
       return m_topSpeed;
    }
    void Save(ostream &) = 0;
};
class WheeledLandVehicle : public VehicleOjbect {
    int m_numberOfWheels
public:
    WheeledLandVehicle(int topSpeed, int numberOfWheels)
    : VechicleObject(topSpeed)
    , m_numberOfWheels(numberOfWheels)
    {}
    int NumberOfWheels() {
      return m_numberOfWheels;
    }
    void Save(ostream &);
};
class TrackedLandVehicle : public VehicleObject {
   int m_numberOfTracks;
public:
   int TrackedLandVehicle (int topSpeed, int numberOfTracks)
   : VechicleObject(topSpeed)
   , m_numberOfTracks (numberOfTracks )
   {}
   int NumberOfTracks() {
      return m_numberOfTracks;
   }
   void Save(ostream &);
 };
```

In this example the VehicleObject is an abstract base class as it has an abstract member function. It is not a pure abstract class as it has both data and concrete member functions. The class WheeledLandVehicle is derivivation 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 VehicleObject class.

This is a bit of a contrived example but it does show how that you can share implementation details among a hierarchy of classes. Each class further refining a concept. This is not always the best way to implement an interface but in some cases it works very well. In general for ease of maintence 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 complile.

Example of useage 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<DrawableOjbect *> 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 geneneral 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 comonality between concrete classes, creation of abstract class which inherites 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 inheritence or aggregation should be used. Too many layers of inheritence can make the maintence and useage of a class difficult. Generally, the maximum accepted layers of inheritence 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.

Stream classes

Streams allow text to be passed in and out of a medium, whether the screen or a file. If it is from a file, the stream must be defined, but the stream for the screen is defined in a header file <iostream> as **cout** and **cin**. Streams that output text like cout must be followed by and separate data by the right stream manipulator, >>. Streams that input text like cin need the left stream manipulator, <<.

The syntax of cout: **cout << "Text which will be displayed." << variable_name;**

More data can be passed by adding another stream manipulator to separate the data.

For example,

```
#include <iostream>
using namespace std;

int main()
{
    int a = 1;
    cout << "Hello world!" << a;
    return 0;
}
```

Result :

```
Hello world!1
```

To add a line break, simplily use **"\n"**, **endl** or **ASCII character no. 13**.

For instance,

```
#include <iostream>
using namespace std;

int main()
{
    int a = 1;
    char x = 13;
    cout << "Hello world!" << "\n" << a << endl << x << endl;
    return 0;
}
```

Result:

```
Hello world!
1
(a blank line)
```

To open a file void open (const char * filename, openmode mode = in | out); where filename is the name of the file to be opened. openmode can be one of these values:

ios::app Leaves the data in the file and adds new data to the end. ios::out Puts new data in the file however, and data allready there is deleted. ios::in Reads data from the file.

Requires #include <fstream>

Example, to open a file called Test.txt and write "HELLO, HOW ARE YOU?" to it:

```
#include <fstream>

using namespace std;

int main()
{
    fstream file;
    file.open("Test.txt", ios::out);
    file << "HELLO, HOW ARE YOU?";
    file.close();
    return 0;
}
```

Advanced Features

Basics

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;
 }
```

Exception hierarchy

You may throw as exception an object (like class 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
  thow InvalidArgumentException();
}
```

```
int main(int argc, const char* argv[]) {
  try
  {
     operate(atoi(argv[0]), atof(argv), atof(argv));
  }
  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. A better approach is:

```
void foo()
{
  throw MyApplicationException();
}
void bar()
{
  try
  {
    foo();
  }
  catch(MyApplicationException& e)
  {
    // Handle exception
  }
}
```

This method has all the advantages - The compiler is responsible for destroying the object, and no copying is done!

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.

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

You can throw an exception from a constructor. You must **never** throw an exception from a destructor! If you *delete* an object, and its destructor throws an exception, the state of the program is undefined!

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:

```
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 creating a puny little templated function, you would rather use templates in a class, so that the class may handle more than one datatype. If you've noticed, pmatrix and pvector are both able to handle creating matrices and vectors of int, double, and etc. This is because there is a line, `template <class SOMETYPE>` in the line preceding the declaration of the class. Just take a look:

```
template <class SOMETYPE>
class pmatrix {
  ...
  ...
  ...
};
```

If you want to declare a function that will return your **TYPEPARAMETER** then replace the return type with your **TYPEPARAMETER** name.

```
template <class SOMETYPE>
SOMETYPE printFunction();
```

Advantages and disadvantages

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

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.

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 supperior to RTTI. As with everything, there are execptions, but the usual rule concerning RTTI is more or less the same as with **goto** statements. Only use it if you have a very good reason to do so, and only if you know what you're doing.

Standard Library

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.

Standard Template Library (STL) offers easy-to-use containers, data types and functions. For example, vector is just like an array, except that vectors size expands to hold more cells.

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-separator 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 == "like");
assert(words == "this");
```

Containers

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

Associative Containers (Sequences)

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 behaviour 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

vector

The **vector** is part of the standard namespace (std::) and as such the class belongs to the **STL** (*Standard Template Libraries*) and is a template class in itself, it is a *Associative 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.

A **vector** would be ideal choice in place of 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).

NOTE:
If you create a vector you can access its data using consecutive pointers:

```
std::vector<type> myvector;
typevar = myvector[0]; etc...
```

this information is present in INCITS/ISO/IEC 14882-2003 but was not properly documented in the 1998 version of the C++ standard but it is expressed in the recent Standard TC.
Watching out for how long that pointer is valid. You should also keep in mind that std::vector<..>::iterator may not be a pointer the safer mode is to simply use the iterator.

vector::Iterators

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 amortized constant time
clear; a member function of the **vector** used to erase its data elements. Note however that if the data elements contained point to memory that was created dynamically (ie: 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**.
back; a member function of the **vector** that returns a reference to the last data element of the **vector**.

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;
```

Linked lists

The STL provides a templated class called **list** (part of the standard namespace (std::)) which implements a doubly-w:linked list. Linked lists can insert or remove elements in the middle in constant time, but do not have random access.

list examples

Associative Containers (key and value)

Associative Containers 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.

- 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 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 class

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. 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 (...) {
  v.erase(i);
}
```

Do:

```
for (...) {
  i = v.erase(i);
}
```

The `erase()` method returns the next valid iterator, or `end()`, thus ending the loop.

I/O Streams

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.

Functors

A functor or function object, is an object that has an `operator ()`.

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

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 `basic_string<>` and its standard specializations `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 of characters including spaces, when reading into a string using cin and the extraction operator (>>), only the those 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

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. An uninitialized string contains nothing, ie. no characters, not even a `NULL`.

```
string str2 = str;
```

Will trigger the copy constructor. `std::string` knows enough to make a deep copy of the characters it stores.

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;
```

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 find = "o";
int position = str.find(find, 0);
std::cout << str[position];
```

Will simply print "o". The value of `pos` is 4, 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 `string::npos`.This simple code searches a string for all occurrences of 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.

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;
}
```

Beyond the C++ Standard: In the real world

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. APIs may provide functions that allow interface with the the hardware or operating system to allow communication between the program and the hardware or operating system. Software Developement Kits or SDK describe the APIs that can be used for program developement.

Compiled libraries consists of two parts:

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

This section will try to include all major libraries that the programmer should know about or have at least a passing idea of what they are. They can be taken as extensions to the standard or just as a life line for surviving on an alien OS API. Don't reinvent the wheel; too much energy has been spent by generations of programmers to write safe and "portable" code.

Binary/Source Compatibility

Libraries comes 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)

Static and Dynamic Libraries

Windows Libraries

Linux/BSD Libraries

MacOS Libraries

Crossplatform or Portable Libraries

Boost Library

Boost library

The **Boost library** provides free peer-reviewed portable C++ source libraries. The emphasis is on libraries which work well with the C++ Standard Library. The libraries are intended to be widely useful, and are in regular use by thousands of programmers across a broad spectrum of applications.

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.

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.

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

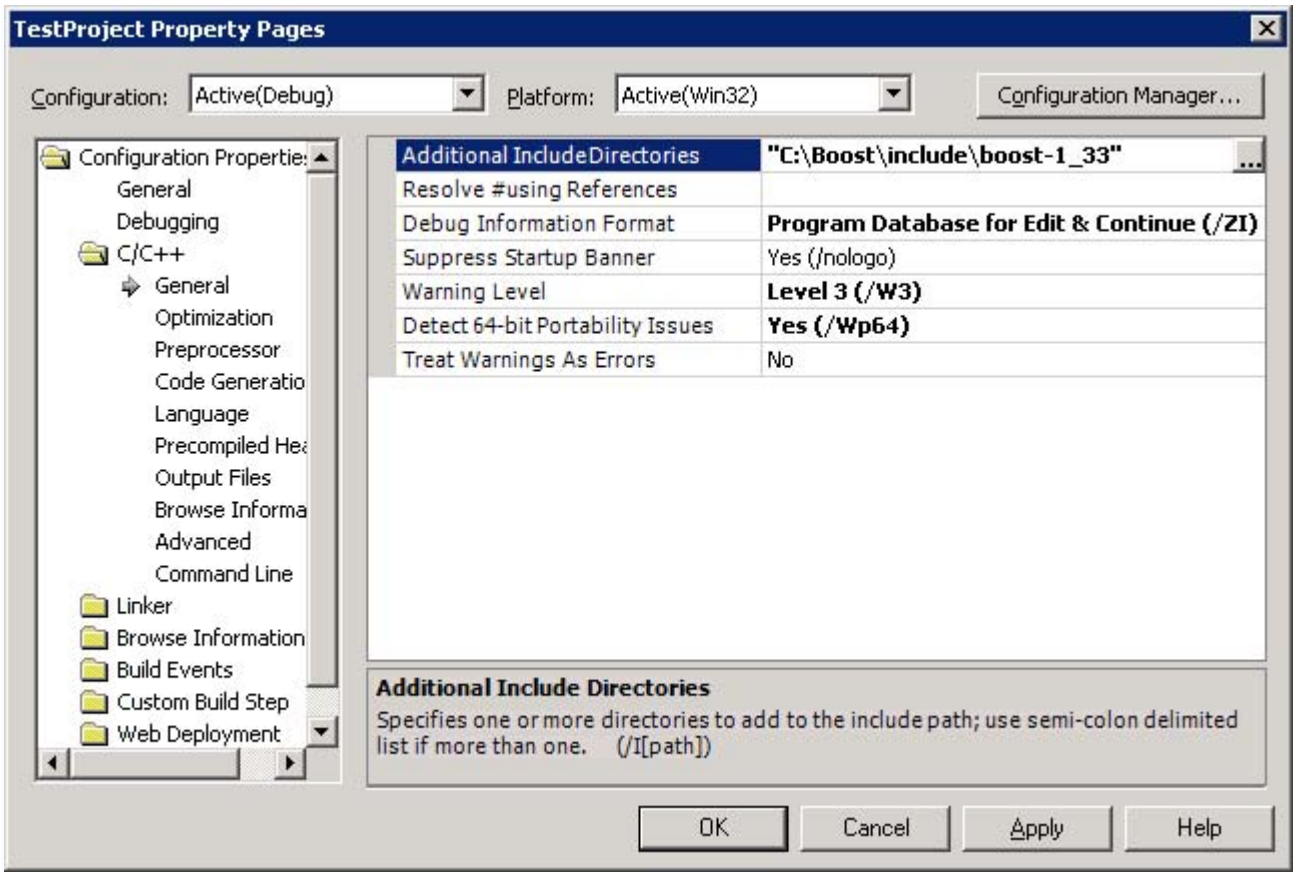
using one codebase. It can be used from languages such as C++, Python, Perl, and C#/.NET. Unlike other cross-platform toolkits, wxWidgets applications look and feel native. This is because wxWidgets uses the platform's own native controls rather than emulating them. It's also extensive, free, open-source, 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.

Procedure to set up Visual Studio

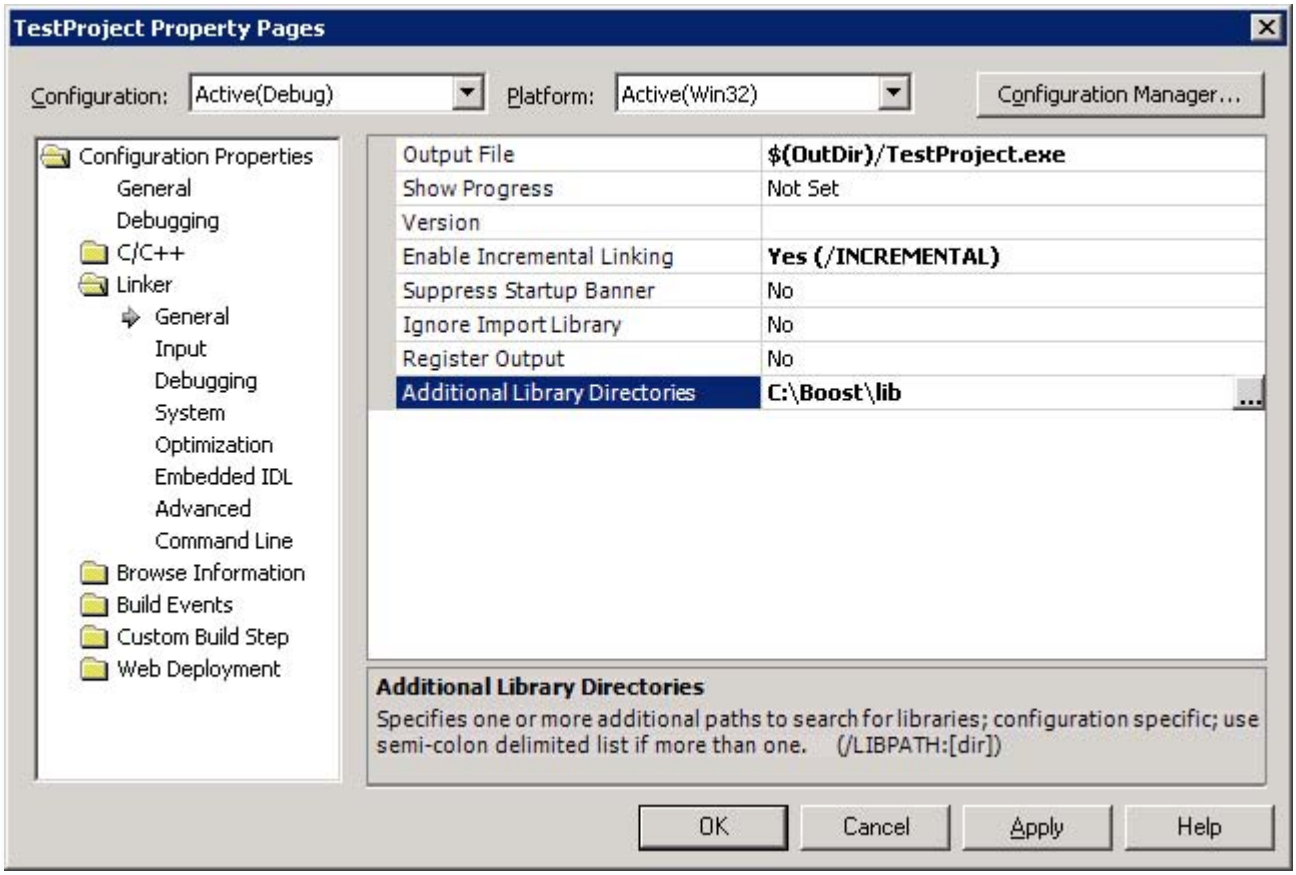
This section serves an an example on how to set up static libraries for usage in Visual Studio. The Boost library serves as an example library.

There are three steps which have to be performed:

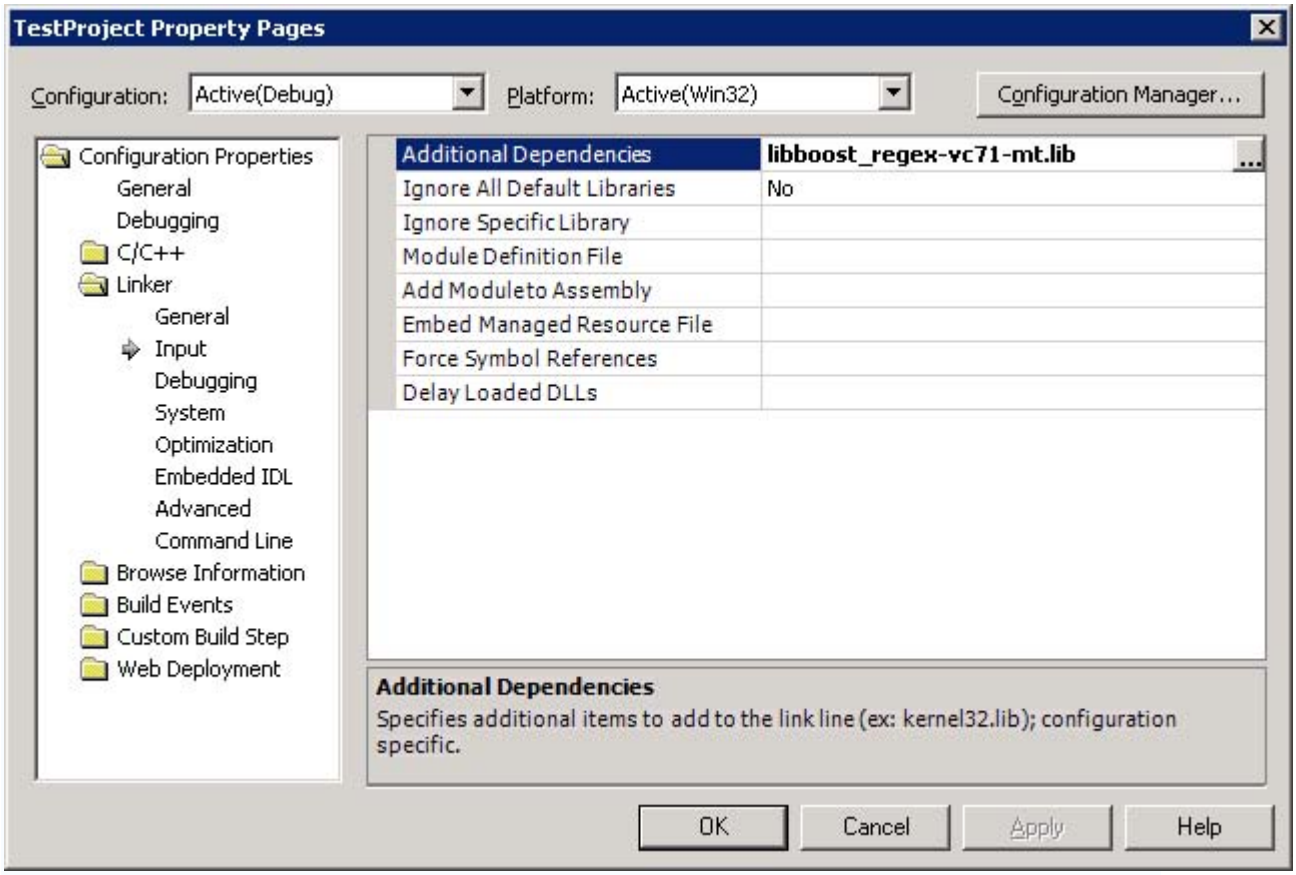
- Set up the *include directory*, which contains the header files of your library



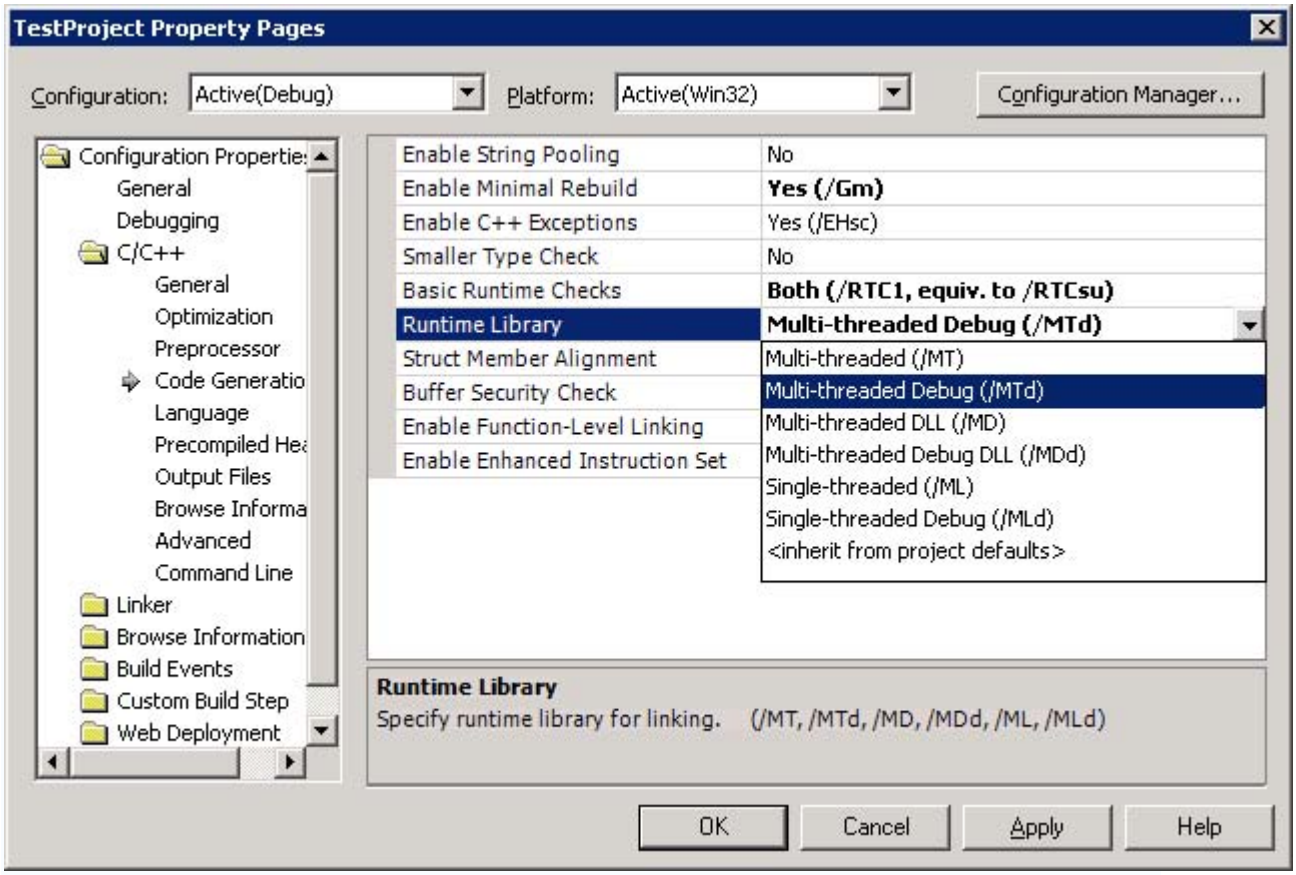
- Set up the *library directory*, which contains the path of the library file(s)



- 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 wether they are compiled for *single- or multithreaded runtime* and/or *debug runtime*, as well as wether they contain *debug symbols*.



Boost Library

Boost library

The **Boost library** (http://www.boost.org/) provides free peer-reviewed portable C++ source libraries. The emphasis is on libraries which work well with the C++ Standard Library. The libraries are intended to be widely useful, and are in regular use by thousands of programmers across a broad spectrum of applications.

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.

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.

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 plaform dependent.

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 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 SDK.

In simple terms the API can be considered as a medium through which the user (user programs) interacts with the operating system to achieve a task (*Software Development Kit*).

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.

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. Operators need not always be symbols.

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.

Here another sample of Operator Overloading

A simple Message Header

```
#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 non-members (the left operand should be the stream - cout << 3, not 3 << cout)
2. the stream must be passed by references (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:

```
ostream& operator<<(ostream& out, const Vector2D& vec) // output
{
  out << "(" << vec.x() << ", " << vec.y() << ")";
  return out;
}
istream& operator>>(istream& in, Vector2D& vec) // input
{
  double x, y;
  in >> x >> y;
  vec.set_x(x);
  vec.set_y(y);
  return in;
}
```

Assignment operator

The assignment operator, =, must be a member function, and is given default behavior for user-defined classes by the compiler, performing an assignment of every member using its assignment operator. This behavior is generally acceptable for simple classes which only contain variables. However, where a class contains references or pointers to outside resources, the assignment operator should be overloaded (as general rule, whenever a destructor and copy constructor are needed so is the assignment operator), otherwise, for example, two strings would share the same buffer and changing one would change the other.

In this case, an assignment operator should perform two duties:

1. clean up the old contents of the object
2. copy the resources of the other object

For classes which contain raw pointers, before doing the assignment, the assignment operator should check for self-assignment, which generally will not work (as when the old contents of the object are erased, they cannot be copied to refill the object). Self

assignment is generally a sign of a coding error, and thus for classes without raw pointers, this check is often omitted, as while the action is wasteful of cpu cycles, it has no other affect 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).

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.

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()

Compound assignment operators

- += (addition-assignment)
- -= (subtraction-assignment)
- *= (multiplication-assignment)

- /= (division-assignment)
- **%**= (modulus-assignment)
- **&**= (AND-assignment)
- |= (OR-assignment)
- ^= (XOR-assignment)
- >>= (shift-right-assignment)
- <<= (shift-left-assignment)

Compound assignment operators should be overloaded as member functions, as they change the left-hand operand. Like all other operators (except basic assignment), compound assignment operators must be explicitly defined, they will not be automatically (e.g. overloading = and + will not automatically overload +=). A compound assignment operator should work as expected: A @= B should be equivalent to A = A @ B. An example of += for a two-dimensional mathematical vector type follows.

```
Vector2D& Vector2D::operator+=(const Vector2D& right)
{
  *this = *this + right;
  return *this;
}
```

Increment and decrement operators

- ++ (increment)
- -- (decrement)

Increment and decrement have two forms, prefix (++i) and postfix (i++). To differentiate, the postfix version takes a dummy integer. Increment and decrement operators are most often member functions, as they generally need access to the private member data in the class. The prefix version in general should return a reference to the changed object. The postfix version should just return a copy of the original value. In a perfect world, A += 1, A = A + 1, A++, ++A should all leave A with the same value.

Example

```
SomeValue& SomeValue::operator++() // prefix
{
  ++data;
  return *this;
}
SomeValue SomeValue::operator++(int unused) // postfix
{
  SomeValue result = *this;
  ++data;
  return result;
}
```

Often one operator is defined in terms of the other for ease in maintenance, especially if the function call is complex.

```
SomeValue SomeValue::operator++(int unused) // postfix
```

```
 {
   SomeValue result = *this;
   ++(*this); // call SomeValue::operator++()
   return result;
 }
```

Subscript operator

The subscript operator, **[ ]**, is a binary operator which must be a member function (hence it takes only one explicit parameter, the index). The subscript operator is not limited to taking an integral index. For instance, the index for the subscript operator for the std::map template is the same as the type of the key, so it may be a string etc. The subscript operator is generally overloaded twice; as a non-constant function (for when elements are altered), and as a constant function (for when elements are only accessed).

Function call operator

The function call operator, **( )**, is generally overloaded to create objects which behave like functions, or for classes that have a primary operation. The function call operator must be a member function, but has no other restrictions - it may be overloaded with any number of parameters of any type, and may return any type. A class may also have several definitions for the function call operator.

Address of, Reference, and Pointer operators

These three operators, operator&(), operator*() and operator->() can be overloaded. In general these operators are only overloaded for smart pointers, or classes which attempt to mimic the behavior of a raw pointer. The pointer operator, operator->() has the additional requirement that the result of the call to that operator, must return a pointer, or a class with an overloaded operator->(). In general A == *&A should be true.

Example

```
class T {
public:
   const memberFunction() const;
};
// forward declaration
class DullSmartReference;
class DullSmartPointer {
private:
  T *m_ptr;
public:
  DullSmartPointer(T *rhs) : m_ptr(rhs) {};
  DulLSmartReference operator*() const {
    return DullSmartReference(*m_ptr);
  }
  T *operator->() const {
    return m_ptr;
  }
};
class DullSmartReference {
```

```
private:
  T *m_ptr;
public:
  DullSmartReference (T &rhs) : m_ptr(&rhs) {}
  DullSmartPointer operator&() const {
    return DullSmartPointer(m_ptr);
  }
  // conversion operator
  operator T { return *m_ptr; }
};
DullSmartPointer dsp(new T);
dsp->memberFunction(); // calls T::memberFunction
T t;
DullSmartReference dsr(t);
dsp = &dsr;
t = dsr; // calls the conversion operator
```

These are extremely simplified examples designed to show how the operators can be overloaded and not the full details of a SmartPointer or SmartReference class. In general you won't want to overload all three of these operators in the same class.

Comma operator

The comma operator,() **,** can be overloaded. The language comma operator has left to right precedence, the operator,() has function call precedence, so be aware that overloading the comma operator has many pitfalls.

Example

```
MyClass operator,(MyClass const &, MyClass const &);
MyClass Function1();
MyClass Function2();
MyClass x = Function1(), Function2();
```

For non overloaded comma operator, the order of execution will be Function1(), Function2(); With the overloaded comma operator, the compiler can call either Function1(), or Function2() first.

Member access operators

The two member access operators, operator->*() and operator.*() can be overloaded. The most common use of overloading these operators is with defining expression template classes, which is not a common programming technique. Clearly by overloading these operators you can create some very unmaintainable code so overload these operators only with great care.

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.

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/) essentialy 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/)

: 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 API conventions

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 usefull 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.

Basics in building "windows"

Window eventhandling

Resources

DLLs

.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.

Registry

IO (Input/Output)

Security

Processes and Threads

Network

Network applications are often built in C++ utilizing the WinSock 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

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.

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 paterns 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 for standardization, with emergence and proliferation of many competing software new design methodologies, concepts, notations, terminologies, processes, and cultures associated with them, the need for unification was self evident by the sheer number of parallel development. 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.

Programming Patterns

Software design patterns are an emerging tool for guiding and documenting system design. This portion of the book, explains when and how various patterns can be used with representative implementations in popular OO languages like C++ and Java.

Creational Patterns

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.

Factory Method

In a situation where a design requires the creation of many different types of objects, all which come from a common base type, then the architect may wish to implement the Factory Design Pattern.

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.

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 mHybernating; // 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 methods, 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

Singleton

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

The term **Singleton** refers to an object that can only be instantiated once. This pattern is generally used where a global variable would have otherwise been used. The main advantage of the singleton is that its existence is guaranteed. Other advantages of the design pattern include the clarity, from the unique access, that the object used is not on the local stack. Some of the downfalls of the object include that, like a global variable, it can be hard to tell what chunk of code corrupted memory, when a bug is found, since everyone has access to it.

Let's take a look at how a Singleton differs from other variable types.

Like a global variable, the Singleton exists outside of the scope of any functions. Traditional implementation uses a static member function of the Singleton class, which will create a single instance of the Singleton class on the first call, and forever return that instance. The following code example illustrates the elements of a C++ singleton class, that simply stores a single string.

```
class StringSingleton
{
public:
    // Some accessor functions for the class, itself
    std::string GetString() const
    {return mString;}
    void SetString(const std::string &newStr)
    {mString = newStr;}

    // The magic function, which allows access to the class from
anywhere
    // To get the value of the instance of the class, call:
    //     StringSingleton::Instance().GetString();
    static StringSingleton &Instance()
    {
        // This line only runs once, thus creating the only instance in
existence
        static StringSingleton *instance = new StringSingleton;
```

```
        // dereferencing the variable here, saves the caller from
having to use
        // the arrow operator, and removes tempation to try and delete
the
        // returned instance.
        return *instance; // always returns the same instance
    }

private:
    // We need to make some given functions private to finish the
definition of the singleton
    StringSingleton(){} // default constructor available only to
members or friends of this class

    // Note that the next two functions are not given bodies, thus any
attempt
    // to call them implicitly will return as compiler errors. This
prevents
    // accidental copying of the only instance of the class.
    StringSingleton(const StringSingleton &old); // disallow copy
constructor
    const StringSingleton &operator=(const StringSingleton &old);
//disallow assignment operator

    // Note that although this should be allowed,
    // some compilers may not implement private destructors
    // This prevents others from deleting our one single instance,
which was otherwise created on the heap
    ~StringSingleton(){}
private: // private data for an instance of this class
    std::string mString;
};
```

Variations of Singletons:

Applications of Singleton Class:

One common use of the singleton design pattern is for application configurations. Configurations may need to be accessible globally, and future expansions to the application configurations may be needed. The subset C's closest alternative would be to create a single global **struct**. This had the lack of clarity as to where this object was instantiated, as well as not guaranteeing the existence of the object.

Take, for example, the situation of another developer using your singleton inside the constructor of their object. Then, yet another developer decides to create an instance of the second class in the global scope. If you had simply used a global variable, the order of linking would then matter. Since your global will be accessed, possibly before main begins executing, there is no definition as to whether the global is initialized, or the constructor of the second class is called first. This behavior can then change with slight modifications to other areas of code, which would change order of global code execution. Such an error can be very hard to debug. But, with use of the singleton, the first time the

object is accessed, the object will also be created. You now have an object which will always exist, in relation to being used, and will never exist if never used.

A second common use of this class is in updating old code to work in a new architecture. Since developers may have used 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

Composite

Decorator

Facade

Flyweight

Proxy

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

Interpreter

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:

```
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
overlaoding
        // 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().

```
ex)

// 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;
```

Mediator

Memento

Observer

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.

Optimization

Definition:

> **Optimization** is the process of fine tunning the end results of an executable program to maximize effiency or to minimize resource usage.

Modern compilers generally have settings to automate the most common optimizations during the compiling process. Often times, greatly reducing the number of optimizations that programmers might need to consider. Modern compilers commonly take advantage of knowledge about a specific system and architecture in the optimization process, as well as years of optimization research results, to further improve performance and minimize resource usage. Such optimizations are best left to the compiler since bloat or performance costs could result when done by hand.

When faced with a problem you try to find solutions that solve the problem. Sometimes the solution is straightforward and at other times the possible solutions aren't as straightforward and may require planning and deciding what steps to take. **Algorithms** are the detailed steps which form the solution to a problem for programmers. Algorithm research continue to find simpler and faster ways to solve problems, reconsider and reduce problems, and divide problems into easier to manage problems. Optimization is very important in algorithmic programming, but not all algorithms may benefit from optimization.

Problems like searching for names in a list or sorting them are solved with algorithms. Sorting algorithms for example, have been developed since the 1950s and new solutions are still being found. The choice of what algorithm to use depends on what is more important time or space. If it is more important that a program be fast than conserve memory usage, often values can be stored ahead of time that will allow for less computation later on. Similarly, an algorithm can use far less space if it is given the time to re-compute all values.

Often, a program can be optimized by simply realizing that it is doing more than it has to. As there are many ways of solving problems, one solution may work but be grossly inefficient when compared with other methods, as was the case with the first sorting algorithm, the bubble sort. Problems that involve searching through lists or binary trees may be exponentially slower than they need to be if a naive approach is used. If a part of the program is too slow to perform reliably, researching the problem at hand for a faster method may be worthwhile.

Redundancies

Redundancies can increase the size and time needed to solve problems. Modern compilers can usualy eliminate some redundancies, such as, variables, functions, values and statements never used and constant variables, values and computational results. However, the complexity of algorithms can reduce the effectiveness of eliminating redundancies by the optimization process of compilers.

- Example of redundancies

```
int foo = 0, bar = 2+2*4, baz = 9;

if (foo) {
  while (foo) {
    ...
  }
}

if (bar) {
  do_loop(bar);
}

void do_loop(int bar) {
  while (bar) {
```

```
    ...
  }
  return;
  do_nothing();
}
```

Common strategy for programmers to decrease or eliminate redundancies:

- when creating small functions assume a valid state.
- limit validating tests to events outside your control such as user input, computer resources and external functions.