UNIT 1

GENESIS OF C++

C++ began as an expanded version of C. The C++ extensions were first invented by Bjarne Stroustrup in 1979 at Bell Laboratories in Murray Hill, New Jersey. He Initially called the new language "C with Classes." However, in 1983 the name was Changed to C++.

Although C was one of the most liked and widely used professional programming Languages in the world, the invention of C++ were necessitated by one major programming factor: increasing complexity. Over the years, computer programs have become larger and more complex. Even though C is an excellent programming language, it has its limits. In C, once a program exceeds from 25,000 to 100,000 lines of code, it becomes so complex that it is difficult to grasp as a totality. The purpose of C++ is to allow this barrier to be broken. The essence of C++ is to allow the programmer to comprehend and manage larger, more complex programs.

Most additions made by Stroustrup to C support object-oriented programming, Sometimes referred to as OOP. Stroustrup states that some of C++'s object-oriented features were inspired by another object-oriented language called Simula67. Therefore, C++ represents the blending of two powerful programming methods.

Since C++ was first invented, it has undergone three major revisions, with each adding to and altering the language. The first revision was in 1985 and the second in 1990. The third occurred during the standardization of C++. Several years ago, work began on a standard for C++. Toward that end, a joint ANSI (American National Standards Institute) and ISO (International Standards Organization) standardization committee was formed. The first draft of the proposed standard was created on January 25, 1994. In that draft, the ANSI/ISO C++ committee kept the features first defined by Stroustrup and added some new ones as well. But in general, this initial draft reflected the state of C++ at the time.

Soon after the completion of the first draft of the C++ standard, an event occurred that caused the language to be greatly expanded: the creation of the Standard Template Library (STL) by Alexander Stepanov. The STL is a set of generic routines that you can use to manipulate data. It is both powerful and elegant, but also quite large. Subsequent to the first draft, the committee voted to include the STL in the specification for C++. The addition of the STL expanded the scope of C++ well beyond its original definition. While important, the inclusion of the STL, among other things, slowed the standardization of C++.
It is fair to say that the standardization of C++ took far longer than anyone had Expected when it began. In the process, many new features were added to the language and many small changes were made. In fact, the version of C++ defined by the C++ committee is much larger and more complex than Stroustrup's original design.

However, the standard is now complete. The final draft was passed out of committee on November 14, 1997. A standard for C++ is now a reality.

Introduction to object oriented programming

C++ was developed by Bjarne Stroustrup of AT&T to add object-oriented Constructs to the C language. Because object-oriented methodology was relatively new at the time, and all existing implementations of object-oriented languages were rather slow and Inefficient (most were interpreters), one goal of C++ was to maintain the efficiency of C.

Since object-oriented programming (OOP) drove the creation of C++, it is necessary to Understand its foundational principles. OOP is a powerful way to approach the job of Programming. Programming methodologies have changed dramatically since the invention of the computer, primarily to accommodate the increasing complexity of programs. For example, when computers were first invented, programming was done by toggling in the binary machine instructions using the computer's front panel. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that a programmer could deal with larger, increasingly complex programs, using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were introduced that gave the programmer more tools with which to handle complexity. The first widespread language was, of course, FORTRAN. Although FORTRAN was a very impressive first step, it is hardly a language that encourages clear, easy-to understand programs.

The 1960s gave birth to structured programming. This is the method encouraged by languages such as C and Pascal. The use of structured languages made it possible to write moderately complex programs fairly easily. Structured languages are characterized by their support for stand-alone subroutines, local variables, rich control constructs, and their lack of reliance upon the GOTO. Although structured languages are a powerful tool, even they reach their limit when a project becomes too large.
Consider this: At each milestone in the development of programming, techniques and tools were created to allow the programmer to deal with increasingly greater complexity. Each step of the way, the new approach took the best elements of the previous methods and moved forward. .
Object-oriented programming took the best ideas of structured programming and combined them with several new concepts. The result was a different way of organizing a program. In the most general sense, a program can be organized in one of two ways: around its code (what is happening) or around its data (who is being affected). Using only structured programming techniques, programs are typically organized around code. This approach can be thought of as "code acting on data." For example, a program written in a structured language such as C is defined by its functions, any of which may operate on any type of data used by the program. Object-oriented programs work the other way around. They are organized around data, with the key principle being "data controlling access to code." In an object-oriented language, you define the data and the routines that are permitted

to act on that data. Thus, a data type defines precisely what sort of operations can be applied to that data.

To support the principles of object-oriented programming, all OOP languages have three traits in common: encapsulation, polymorphism, and inheritance. Let's examine each.

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. In an object-oriented Language, code and data may be combined in such a way that a self-contained "black box" is created. When code and data are linked together in this fashion, an *object* is created. In other words, an object is the device that supports encapsulation.
Within an object, code, data, or both may be *private* to that object or *public*. Private code or data is known to and accessible only by another part of the object. That is, private code or data may not be accessed by a piece of the program that exists outside the object. When code or data is public, other parts of your program may access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object.
For all intents and purposes, an object is a variable of a user-defined type. It may seem strange that an object that links both code and data can be thought of as a variable. However, in object-oriented programming, this is precisely the case. Each time you define a new type of object, you are creating a new data type. Each specific instance of this data type is a compound variable.

Polymorphism

Object-oriented programming languages support *polymorphism*, which is characterized by the phrase "one interface, multiple methods." In simple terms, polymorphism is the attribute that allows one interface to control access to a general class of actions. The specific action selected is determined by the exact nature of the situation. A real-world example of polymorphism is a thermostat. No matter what type of furnace your house has (gas, oil, electric, etc.), the thermostat works the same way. In this case, the thermostat (which is the interface) is the same no matter what type of furnace (method) you have. For example, you might have a program that defines three different types of stacks. One stack is used for integer values, one for character values, and one for floating-point values. Because of polymorphism, you can define one set of names, **push()** and **pop()** , that can be used for all three stacks. In your program you will create three specific versions of these functions, one for each type of stack, but names of the functions will be the same. The compiler will automatically select the right function based upon the data being stored.
Thus, the interface to a stack—the functions **push()** and **pop()** —are the same no matter which type of stack is being used. The individual versions of these functions define the specific implementations (methods) for each type of data.
Polymorphism helps reduce complexity by allowing the same interface to be used to access a general class of actions. It is the compiler's job to select the *specific action* (i.e., method) as it applies to each situation. You, the programmer, don't need to do

this selection manually. You need only remember and utilize the *general interface*. The first object-oriented programming languages were interpreters, so polymorphism was, of course, supported at run time. However, C++ is a compiled language. Therefore, in C++, both run-time and compile-time polymorphism are supported.

Inheritance

Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of *classification*. If you think about it, most knowledge is made manageable by hierarchical classifications. For example, a Red Delicious apple is part of the classification *apple*, which in turn is part of the *fruit* class, which is under the larger class *food*. Without the use of classifications, each object would have to define explicitly all of its characteristics. However, through the use of classifications, an object need only define those qualities that make it unique within its class. It is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. As you will see, inheritance is an important aspect of object-oriented programming.

Data abstraction

Abstraction refes to the act of representing essential features without including background details or explanations.classes use the concept of abstraction and are defined as a list of abstract attributes and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called data members because they hold information. The functions that operate on these data are sometimes called methods or member functions.

A Sample C++ Program

Let's start with the short sample C++ program shown here.

```
#include <iostream>
using namespace std;
int main()
{
int i;

cout << "This is output.\n";   // this is a single line comment

                    /* we can still use C style comments */
// input a number using >>
```

```
cout << "Enter a number: ";
cin >> i;
// now, output a number using <<

cout << i << " squared is " << i*i << "\n";

return 0;
}
```

As you can see, this program looks much different from the C subset programs found in Part One. A line-by-line commentary will be useful. To begin, the header **<iostream>** is included. This header supports C++-style I/O operations. (**<iostream>** is to C++ what **stdio.h** is to C.) Notice one other thing: there is no **.h** extension to the name **iostream**. The reason is that **<iostream>** is one of the new-style headers defined by Standard C++. New-style headers do not use the **.h** extension.

The next line in the program is

```
using namespace std;
```

This tells the compiler to use the **std** namespace. Namespaces are a recent addition to C++. A namespace creates a declarative region in which various program elements can be placed. Namespaces help in the organization of large programs. The **using** statement informs the compiler that you want to use the **std** namespace. This is the namespace in which the entire Standard C++ library is declared. By using the **std**
namespace you simplify access to the standard library. The programs in Part One, which use only the C subset, don't need a namespace statement because the C library functions are also available in the default, global namespace.

Since both new-style headers and namespaces are recent additions to C++, you may encounter older code that does not use them. Also, if you are using an older compiler, it may not support them. Instructions for using an older compiler are found later in this chapter.

Now examine the following line.

```
int main()
```

Notice that the parameter list in **main()** is empty.

In C++, this indicates that **main()** has no parameters. This differs from C. In C, a function that has no parameters must use **void** in its parameter list, as shown here:

```
int main(void)
```

This was the way **main()** was declared in the programs in Part One. However, in C++, the use of **void** is redundant and unnecessary. As a general rule, in C++ when a function takes no parameters, its parameter list is simply empty; the use of **void** is not required.

The next line contains two C++ features.

cout << "This is output.\n"; // this is a single line comment

First, the statement
cout << "This is output.\n";
causes **This is output**. to be displayed on the screen, followed by a carriage return line feed combination. In C++, the << has an expanded role. It is still the left shift operator, but when it is used as shown in this example, it is also an *output operator*. The word **cout** is an identifier that is linked to the screen.

(Actually, like C, C++ supports I/O redirection, but for the sake of discussion, assume that **cout** refers to the screen.)

You can use **cout** and the << to output any of the built-in data types, as well as strings of characters.

Note that you can still use **printf()** or any other of C's I/O functions in a C++
program. However, most programmers feel that using << is more in the spirit of C++.

Further, while using **printf()** to output a string is virtually equivalent to using << in this case, the C++ I/O system can be expanded to perform operations on objects that you define (something that you cannot do using **printf()**).

What follows the output expression is a C++ *single-line comment*.

C++ defines two types of comments. First, you may use a C-like comment, which works the same in C++ as in C. You can also define a single-line comment by using // ;whatever follows such a comment is ignored by the compiler until the end of the line is reached. In general, C++ programmers use C-like comments when a multiline comment is being created and use C++ single-line comments when only a single-line remark is needed. Next, the program prompts the user for a number. The number is read from the keyboard with this statement:

cin >> i;

In C++, the >> operator still retains its right shift meaning. However, when used as shown, it also is C++'s *input operator*. This statement causes **i** to be given a value read from the keyboard. The identifier **cin** refers to the standard input device, which isusually the keyboard. In general, you can use **cin** >> to input a variable of any of the basic data types plus strings.

The line of code just described is not misprinted. Specifically, there is not supposed to be an ***&*** *in front of the* ***i****. When inputting information using a C-based function like* ***scanf( ),*** *you have to explicitly pass a pointer to the variable that will receive the information. This means preceding the variable name with the "address of" operator,* ***&****. However, because of the way the >> operator is implemented in C++, you do not need (in fact, must not use) the* ***&****.* Although it is not illustrated by the example, you are free to use any of the C-based input functions, such as **scanf(),** instead of using >>.However, as with **cout**, most programmers feel that **cin** >> is more in the spirit of C++.

Another interesting line in the program is shown here:
cout << i << "squared is " << i*i << "\n";

Assuming that **i** has the value 10, this statement causes the phrase **10 squared is 100** to be displayed, followed by a carriage return-linefeed. As this line illustrates, you can run together several **<<** output operations.

The program ends with this statement:
return 0;

This causes zero to be returned to the calling process (which is usually the operating system). This works the same in C++ as it does in C. Returning zero indicates that the program terminated normally. Abnormal program termination should be signaled by returning a nonzero value. You may also use the values **EXIT_SUCCESS** and **EXIT_FAILURE** if you like.

Declaring Local Variables
In C++ you may declare local variables at any point within a block—not just at the beginning. Here is an example.

```
#include <iostream>
using namespace std;
int main()
{
float f;
double d;
cout << "Enter two floating point numbers: ";
cin >> f >> d;
cout << "Enter a string: ";
char str[80];   // str declared here, just before 1st use
cin >> str;
cout << f << " " << d << " " << str;
return 0;
}
```

Whether you declare all variables at the start of a block or at the point of first use is completely up to you. Since much of the philosophy behind C++ is the encapsulation of code and data, it makes sense that you can declare variables close to where they are used instead of just at the beginning of the block.

The New C++ Headers

As you know, when you use a library function in a program, you must include its header file. This is done using the **#include** statement. For example, in C, to include the

header file for the I/O functions, you include **stdio.h** with a statement like this:

#include <stdio.h>

Here, **stdio.h** is the name of the file used by the I/O functions, and the preceding statement causes that file to be included in your program. The key point is that this **#include** statement *includes a file*.

When C++ was first invented and for several years after that, it used the same style of headers as did C. That is, it used *header files*. In fact, Standard C++ still supports C-style headers for header files that you create and for backward compatibility.

However, Standard C++ created a new kind of header that is used by the Standard C++ library. The new-style headers *do not* specify filenames. Instead, they simply specify standard identifiers that may be mapped to files by the compiler, although they need not be. The new-style C++ headers are an abstraction that simply guarantee that the appropriate prototypes and definitions required by the C++ library have been declared.

Since the new-style headers are not filenames, they do not have a **.h** extension. They consist solely of the header name contained between angle brackets. For example, here are some of the new-style headers supported by Standard C++.

<iostream> <fstream> <vector> <string>

The new-style headers are included using the **#include** statement. The only difference is that the new-style headers do not necessarily represent filenames.
Because C++ includes the entire C function library, it still supports the standard C-style header files associated with that library. That is, header files such as **stdio.h** or **ctype.h** are still available. However, Standard C++ also defines new-style headers that you can use in place of these header files. The C++ versions of the C standard headers simply add a "c" prefix to the filename and drop the **.h**. For example, the C++ new-style header for **math.h** is **<cmath>**. The one for **string.h** is **<cstring>**.
Although it is currently permissible to include a C-style header file when using C library functions,this approach is deprecated by Standard C++ (that is, it is not recommended).

If your compiler does not support new-style headers for the C function library, then simply substitute the old-style, C-like headers.Since the new-style header is a recent addition to C++, you will still find many,many older programs that don't use it. These programs employ C-style headers, in which a filename is specified. As the old-style skeletal program shows, the traditional way to include the I/O header is as shown here.

#include <iostream.h>

This causes the file **iostream.h** to be included in your program. In general, an old-style header file will use the same name as its corresponding new-style header with

a **.h** appended.
As of this writing, all C++ compilers support the old-style headers. However, the old-style headers have been declared obsolete and their use in new programs is not recommended.

Namespaces

When you include a new-style header in your program, the contents of that header are contained in the **std** namespace. A *namespace* is simply a declarative region. The purpose of a namespace is to localize the names of identifiers to avoid name collisions. Elements declared in one namespace are separate from elements declared in another.

Originally, the names of the C++ library functions, etc., were simply put into the global namespace (as they are in C). However, with the advent of the new-style headers, the contents of these headers were placed in the **std** namespace.
the statement

using namespace std;

brings the **std** namespace into visibility (i.e., it puts **std** into the global namespace).

After this statement has been compiled, there is no difference between working with an old-style header and a new-style one.

One other point: for the sake of compatibility, when a C++ program includes a C header, such as **stdio.h**, its contents are put into the global namespace. This allows a C++ compiler to compile C-subset programs.

Concept of Classes and objects

A *class* is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.
An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.
Classes are generally declared using the keyword class, with the following format:

```
class class_name {
access_specifier_1:
member1;
access_specifier_2:
member2;
```

...
} object_names;

Where class_name is a valid identifier for the class, object_names is an optional list of names for objects of this class. The body of the declaration can contain members, that can be either data or function declarations, and
optionally access specifiers.

All is very similar to the declaration on data structures, except that we can now include also functions and members, but also this new thing called *access specifier*. An access specifier is one of the following three
keywords: private, public or protected. These specifiers modify the access rights that the members following them acquire:

• private members of a class are accessible only from within other members of the same class or from their *friends*.
• protected members are accessible from members of their same class and from their friends, but also from members of their derived classes.
• Finally, public members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members.

Therefore, any member that is declared before one other class specifier automatically has private access. Forexample:

```
class CRectangle {
int x, y;
public:
void set_values (int,int);
int area (void);
} rect;
```

Declares a class (i.e., a type) called CRectangle and an object (i.e., a variable) of this class called rect. This class contains four members: two data members of type int (member x and member y) with private access (because
private is the default access level) and two member functions with public access: set_values() and area(), ofwhich for now we have only included their declaration, not their definition.

Notice the difference between the class name and the object name: It is the same relationship int and a have in the following declaration:

```
int a;
```

where int is the type name (the class) and a is the variable name (the object).

After the previous declarations of CRectangle and rect, we can refer within the body of the program to any of the public members of the object rect as if they were normal functions or normal variables, just by putting the object's
name followed by a dot (.) and then the name of the member. All very similar to what we did with plain data structures before.

For example:
rect.set_values (3,4);
myarea = rect.area();

The only members of rect that we cannot access from the body of our program outside the class are x and y, since they have private access and they can only be referred from within other members of that same class.

Here is the complete example of class CRectangle:

```
// classes example

#include <iostream>
using namespace std;

class CRectangle
{
int x, y;
public:
void set_values (int,int);
int area () {return (x*y);}
};

void CRectangle::set_values (int a, int b)
 {
x = a;
y = b;
}

int main ()
{
CRectangle rect;
rect.set_values (3,4);
cout << "area: " << rect.area();
return 0;
}
```

area: 12

The most important new thing in this code is the operator of scope (::, two colons) included in the definition of set_values(). It is used to define a member of a class from outside the class definition itself.

You may notice that the definition of the member function area() has been included directly within the definition of the CRectangle class given its extreme simplicity, whereas set_values() has only its prototype declared within
the class, but its definition is outside it. In this outside declaration, we must use the operator of scope (::) to specify that we are defining a function that is a member of the class CRectangle and not a regular global function. The scope operator **(::)** specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, in the function set_values() of the previous code, we have been able to use the variables x and y, which are private members of class CRectangle, which means they are only accessible from other members of their class.

The only difference between defining a class member function completely within its class or to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline
member function by the compiler, while in the second it will be a normal (not-inline) class member function, which in fact supposes no difference in behavior.

Members x and y have private access (remember that if nothing else is said, all members of a class defined with keyword class have private access). By declaring them private we deny access to them from anywhere outside the
class. This makes sense, since we have already defined a member function to set values for those members within the object: the member function set_values(). Therefore, the rest of the program does not need to have direct
access to them. Perhaps in a so simple example as this, it is difficult to see an utility in protecting those two variables, but in greater projects it may be very important that values cannot be modified in an unexpected way
(unexpected from the point of view of the object).

One of the greater advantages of a class is that, as any other type, we can declare several objects of it.

For example, following with the previous example of class CRectangle, we could have declared the object rectb in addition to the object rect:
e.g CRectangle rect,rectb;

UNIT 2

Control Structures in c++

Control Structures

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done
by our program, when and under which circumstances.

With the introduction of control structures we are going to have to introduce a new concept: the *compound statement*
or *block.*
A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces: { }:

{ statement1; statement2; statement3; }

Most of the control structures that we will see in this section require a generic statement as part of its syntax. A statement can be either a simple statement (a simple instruction ending with a semicolon) or a compound
statement (several instructions grouped in a block), like the one just described. In the case that we want the statement to be a simple statement, we do not need to enclose it in braces ({}).
But in the case that we want the statement to be a compound statement it must be enclosed between braces ({}), forming a block.

Conditional structure: if and else

The if keyword is used to execute a statement or block only if a condition is fulfilled. Its form is:
if (condition) statement

Where condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is ignored (not executed) and the program continues right after this conditional structure.

For example,
the following code fragment prints x is 100 only if the value stored in the x variable is indeed 100:

```
if (x == 100)
cout << "x is 100";
```

If we want more than a single statement to be executed in case that the condition is true we can specify a block
using braces { }:

```
if (x == 100)
{
cout << "x is ";
cout << x;
}
```

We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword else. Its form used in conjunction with if is:

if (condition) statement1 else statement2

For example:

```
if (x == 100)
cout << "x is 100";
else
cout << "x is not 100";
```

prints on the screen x is 100 if indeed x has a value of 100, but if it has not -and only if not- it prints out x is not 100.

The if + else structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling if the value currently stored in x is positive, negative or none of them (i.e. zero):

```
if (x > 0)
cout << "x is positive";
else if (x < 0)
cout << "x is negative";
else
```

```
cout << "x is 0";
```

Remember that in case that we want more than a single statement to be executed, we must group them in a block
by enclosing them in braces { }.

Iteration structures (loops)

Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

The while loop

Its format is:

while (expression) statement

and its functionality is simply to repeat statement while the condition set in expression is true.

For example, we are going to make a program to countdown using a while-loop:

```
// custom countdown using while
#include <iostream>
using namespace std;

int main ()
{
int n;
cout << "Enter the starting number > ";
cin >> n;

while (n>0)
{
cout << n << ", ";
--n;
}

cout << "FIRE!\n";
return 0;
}
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

When the program starts the user is prompted to insert a starting number for the countdown. Then the while loop begins, if the value entered by the user fulfills the condition n>0 (that n is

greater than zero) the block that follows the condition will be executed and repeated while the condition (n>0) remains being true.

The whole process of the previous program can be interpreted according to the following script (beginning in main):

1. User assigns a value to n
2. The while condition is checked (n>0). At this point there are two posibilities:
* condition is true: statement is executed (to step 3)
* condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:
cout << n << ", ";
--n;
(prints the value of n on the screen and decreases n by 1)
4. End of block. Return automatically to step 2
5. Continue the program right after the block: print FIRE! and end program.
When creating a while-loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will
continue looping forever. In this case we have included --n; that decreases the value of the variable that is being evaluated in the condition (n) by one - this will eventually make the condition (n>0) to become false after a certain
number of loop iterations: to be more specific, when n becomes 0, that is where our while-loop and our countdown end.

Of course this is such a simple action for our computer that the whole countdown is performed instantly without any practical delay between numbers.

The do-while loop
Its format is:

do statement while (condition);

Its functionality is exactly the same as the while loop, except that condition in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of statement even if condition is
never fulfilled. For example, the following example program echoes any number you enter until you enter 0.

```
// number echoer
#include <iostream>
using namespace std;

int main ()
{
unsigned long n;
```

```
do {
cout << "Enter number (0 to end): ";
cin >> n;
cout << "You entered: " << n << "\n";
} while (n != 0);

return 0;
}
```

```
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

The do-while loop is usually used when the condition that has to determine the end of the loop is determined within the loop statement itself, like in the previous case, where the user input within the block is what is used to
determine if the loop has to end. In fact if you never enter the value 0 in the previous example you can be prompted for more numbers forever.

The for loop
Its format is: **for (initialization; co ndition; increase) statement;**

and its main function is to repeat statement while condition remains true, like the while loop. But in addition, the for loop provides specific locations to contain an initialization statement and an increase statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.

It works in the following way:
1. initialization is executed. Generally it is an initial value setting for a counter variable. This is executed
only once.
2. condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped
(not executed).
3. statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.
4. finally, whatever is specified in the increase field is executed and the loop gets back to step 2.
Here is an example of countdown using a for loop:

```
// countdown using a for loop
#include <iostream>
using namespace std;

int main ()
```

```
{
for (int n=10; n>0; n--)
 {
cout << n << ", ";
}
cout << "FIRE!\n";
return 0;
}
```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!

The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: for (;n<10;) if we wanted to specify no initialization and no increase; or for (;n<10;n++) if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before).
Optionally, using the comma operator (,) we can specify more than one expression in any of the fields included in a for loop, like in initialization, for example. The comma operator (,) is an expression separator, it serves to
separate more than one expression where only one is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
// whatever here...
}
```

This loop will execute for 50 times if neither n or i are modified within the loop:
n starts with a value of 0, and i with 100, the condition is n!=i (that n is not equal to i). Because n is increased by one and i decreased by one, the loop's condition will become false after the 50th loop, when both n and i will be
equal to 50.

Jump statements.

The break statement

Using break we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before its
natural end (maybe because of an engine check failure?):

```
// break loop example
#include <iostream>
using namespace std;

int main ()
{
int n;
for (n=10; n>0; n--)
{
cout << n << ", ";
```

```
if (n==3)
{
cout << "countdown aborted!";
break;
}
}
return 0;
}
```
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!

The continue statement

The continue statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, we are
going to skip the number 5 in our countdown:

```
// continue loop example

#include <iostream>
using namespace std;

int main ()
{
for (int n=10; n>0; n--)
{
if (n==5) continue;
cout << n << ", ";
}
cout << "FIRE!\n";
return 0;
}
```
10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!

The goto statement

goto allows to make an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations.
The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (:).

Generally speaking, this instruction has no concrete use in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using goto:

```
// goto loop example
```

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

int main ()
{
int n=10;
loop:
cout << n << ", ";
n--;
if (n>0) goto loop;
cout << "FIRE!\n";
return 0;
}
```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!

The selective structure: switch.

The syntax of the switch statement is a bit peculiar. Its objective is to check several possible constant values for an
expression. Something similar to what we did at the beginning of this section with the concatenation of several if
and else if instructions. Its form is the following:

```
switch (expression)
{
case constant1:
group of statements 1;
break;
case constant2:
group of statements 2;
break;
.
.
.
default:
default group of statements
}
```
It works in the following way: switch evaluates expression and checks if it is equivalent to constant1, if it is, it executes group of statements 1 until it finds the break statement. When it finds this break statement the program jumps to the end of the switch selective structure.

If expression was not equal to constant1 it will be checked against constant2. If it is equal to this, it will execute group of statements 2 until a break keyword is found, and then will jump to the end of the switch selective structure.

Finally, if the value of expression did not match any of the previously specified constants (you can include as many case labels as values you want to check), the program will execute the statements included after the default: label, if it exists (since it is optional).

Both of the following code fragments have the same behavior:

switch example if-else equivalent

```
switch (x) {
case 1:
cout << "x is 1";
break;
case 2:
cout << "x is 2";
break;
default:
cout << "value of x unknown";
}
if (x == 1) {
cout << "x is 1";
}
else if (x == 2) {
cout << "x is 2";
}
else {
cout << "value of x unknown";
}
```

The switch statement is a bit peculiar within the C++ language because it uses labels instead of blocks. This forces us to put break statements after the group of statements that we want to be executed for a specific condition. Otherwise the remainder statements -including those corresponding to other labels- will also be executed until the end of the switch selective block or a break statement is reached.

For example, if we did not include a break statement after the first group for case one, the program will not automatically jump to the end of the switch selective block and it would continue executing the rest of statements
until it reaches either a break instruction or the end of the switch selective block. This makes unnecessary to include braces { } surrounding the statements for each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression being evaluated. For example:

```
switch (x) {
case 1:
case 2:
case 3:
cout << "x is 1, 2 or 3";
break;
```

```
default:
cout << "x is not 1, 2 nor 3";
}
```

Notice that switch can only be used to compare an expression against constants. Therefore we cannot put variables as labels (for example case n: where n is a variable) or ranges (case (1..3):) because they are not valid C++
constants.

Operators in c++

Once we know of the existence of variables and constants, we can begin to operate with them. For that purpose,
C++ integrates operators. Unlike other languages whose operators are mainly keywords, operators in C++ are mostly made of signs that are not part of the alphabet but are available in all keyboards. This makes C++ code
shorter and more international, since it relies less on English words, but requires a little of learning effort in the beginning.
You do not have to memorize all the content of this page. Most details are only provided to serve as a later reference in case you need it.

Assignment (=)

The assignment operator assigns a value to a variable.
a = 5; This statement assigns the integer value 5 to the variable a. The part at the left of the assignment operator (=) is known as the *lvalue* (left value) and the right one as the *rvalue* (right value). The lvalue has to be a variable
whereas the rvalue can be either a constant, a variable, the result of an operation or any combination of these.

The most important rule when assigning is the *right-to-left* rule: The assignment operation always takes place from right to left, and never the other way:
a = b;
This statement assigns to variable a (the lvalue) the value contained in variable b (the rvalue). The value that was stored until this moment in a is not considered at all in this operation, and in fact that value is lost.

Consider also that we are only assigning the value of b to a at the moment of the assignment operation. Therefore a later change of b will not affect the new value of a.

For example, let us have a look at the following code - I have included the evolution of the content stored in the variables as comments:

```
// assignment operator

#include <iostream>
```

```
using namespace std;
int main ()
{
int a, b; // a:?, b:?
a = 10; // a:10, b:?
b = 4; // a:10, b:4
a = b; // a:4, b:4
b = 7; // a:4, b:7
cout << "a:";
cout << a;
cout << " b:";
cout << b;
return 0;
}
a:4 b:7
```

This code will give us as result that the value contained in a is 4 and the one contained in b is 7. Notice how a was not affected by the final modification of b, even though we declared a = b earlier (that is because of the *right-to leftrule*).

A property that C++ has over other programming languages is that the assignment operation can be used as the rvalue (or part of an rvalue) for another assignment operation.
For example:

```
a = 2 + (b = 5);
```

is equivalent to:

```
b = 5;
a = 2 + b;
```

that means: first assign 5 to variable b and then assign to a the value 2 plus the result of the previous assignment of b (i.e. 5), leaving a with a final value of 7.

The following expression is also valid in C++:

```
a = b = c = 5;
```

It assigns 5 to the all the three variables: a, b and c.

Arithmetic operators (+, -, *, /, %)

The five arithmetical operations supported by the C++ language are:
+ addition – subtraction * multiplication / division % modulo Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see is *modulo*; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. For example, if we write:

```
a = 11 % 3;
```

the variable a will contain the value 2, since 2 is the remainder from dividing 11 between 3.

Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=,^=, |=)

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators:

expression is equivalent to

value += increase; value = value + increase;
a -= 5; a = a - 5;
a /= b; a = a / b;
price *= units + 1; price = price * (units + 1);
and the same for all other operators.

For example:

```
// compound assignment operators
#include <iostream>
using namespace std;
int main ()
{
int a, b=3;
a = b;
a+=2; // equivalent to a=a+2
cout << a;
return 0;
}
```

Increase and decrease (++, --)

Shortening even more some expressions, the increase operator (++) and the decrease operator (--) increase orreduce by one the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus:

```
c++;
c+=1;
c=c+1;
```

are all equivalent in its functionality: the three of them increase by one the value of c.

In the early C compilers, the three previous expressions probably produced different executable code depending on which one was used. Nowadays, this type of code optimization is generally done automatically by the compiler,
thus the three expressions should produce exactly the same executable code.

A characteristic of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable identifier (++a) or after it (a++).

Although in simple expressions like a++ or ++a both have exactly the same meaning, in other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (++a) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a suffix (a++) the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression. Notice the difference:

Example 1 Example 2

```
B=3;
A=++B;
// A contains 4, B contains 4
B=3;
A=B++;
// A contains 3, B contains 4
```

In Example 1, B is increased before its value is copied to A. While in Example 2, the value of B is copied to A andthen B is increased.

Relational and equality operators (==, !=, >, <, >=, <=)

In order to evaluate a comparison between two expressions we can use the relational and equality operators. The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result.

We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other is. Here is a list of the relational and equality operators that can be used in C++:

== Equal to
!= Not equal to
> Greater than
< Less than
>= Greater than or equal to
<= Less than or equal to

Here there are some examples:

```
(7 == 5) // evaluates to false.
(5 > 4) // evaluates to true.
(3 != 2) // evaluates to true.
(6 >= 6) // evaluates to true.
(5 < 5) // evaluates to false.
```

Of course, instead of using only numeric constants, we can use any valid expression, including variables. Suppose
that a=2, b=3 and c=6,

```
(a == 5) // evaluates to false since a is not equal to 5.
(a*b >= c) // evaluates to true since (2*3 >= 6) is true.
(b+4 > a*c) // evaluates to false since (3+4 > 2*6) is false.
```

((b=2) == a) // evaluates to true.

Be careful! The operator = (one equal sign) is not the same as the operator == (two equal signs), the first one is anassignment operator (assigns the value at its right to the variable at its left) and the other one (==) is the equality operator that compares whether both expressions in the two sides of it are equal to each other. Thus, in the last expression ((b=2) == a), we first assigned the value 2 to b and then we compared it to a, that also stores the
value 2, so the result of the operation is true.

Logical operators (!, &&, ||)

The Operator ! is the C++ operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its
operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. For example:

!(5 == 5) // evaluates to false because the expression at its right (5 == 5) is true.
!(6 <= 4) // evaluates to true because (6 <= 4) would be false.
!true // evaluates to false
!false // evaluates to true.

The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise.

The operator || corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. Here are the possible results of a||b:

|| OPERATOR
a b a || b
true true true
true false true
false true true
false false false

For example:
((5 == 5) && (3 > 6)) // evaluates to false (true && false).
((5 == 5) || (3 > 6)) // evaluates to true (true || false).

Conditional operator (?)

The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:
condition ? result1 : result2

If condition is true the expression will return result1, if it is not it will return result2.

```
7==5 ? 4 : 3 // returns 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3 // returns 4, since 7 is equal to 5+2.
5>3 ? a : b // returns the value of a, since 5 is greater than 3.
a>b ? a : b // returns whichever is greater, a or b.
// conditional operator
```

```
#include <iostream>
using namespace std;
int main ()
{
int a,b,c;
a=2;
b=7;
c = (a>b) ? a : b;
cout << c;
return 0;
}
```

In this example a was 2 and b was 7, so the expression being evaluated (a>b) was not true, thus the first value specified after the question mark was discarded in favor of the second value (the one after the colon) which was b,
with a value of 7.

Comma operator (,)
The comma operator (,) is used to separate two or more expressions that are included where only one expressionis expected. When the set of expressions has to be evaluated for a value, only the rightmost expression is
considered.

For example, the following code:

```
a = (b=3, b+2);
```

Would first assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

Bitwise Operators (&, |, ^, ~, <<, >>)

Bitwise operators modify variables considering the bit patterns that represent the values they store.

operator asm equivalent description

& AND Bitwise AND
| OR Bitwise Inclusive OR
^ XOR Bitwise Exclusive OR
~ NOT Unary complement (bit inversion)
<< SHL Shift Left
>> SHR Shift Right

Explicit type casting operator

Type casting operators allow you to convert a datum of a given type to another. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be
converted by the new type enclosed between parentheses (()):

```
int i;
float f = 3.14;
i = (int) f;
```

The previous code converts the float number 3.14 to an integer value (3), the remainder is lost. Here, the typecasting operator was (int). Another way to do the same thing in C++ is using the functional notation:
preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int ( f );
```

Both ways of type casting are valid in C++

sizeof()

This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

```
a = sizeof (char);
```

This will assign the value 1 to a because char is a one-byte long type.
The value returned by sizeof is a constant, so it is always determined before program execution.

UNIT 3

Classes and objects

Classes

Classes are created using the keyword **class**. A class declaration defines a new type that links code and data. This new type is then used to declare objects of that class. Thus, a class is a logical abstraction, but an object has physical existence. In other words, an object is an *instance* of a class.

A class declaration is similar syntactically to a structure . Here is the general form of a **class** declaration that does not inherit any other class.

class *class-name {*
private data and functions
access-specifier:

data and functions
access-specifier:
data and functions
// ...
access-specifier:
data and functions
} object-list;

The *object-list* is optional. If present, it declares objects of the class. Here, ***access-specifier*** is one of these three C++ keywords:

public
private
protected

By default, functions and data declared within a class are private to that class and may be accessed only by other members of the class. The **public** access specifier allows functions or data to be accessible to other parts of your program. The **protected** access specifier is needed only when inheritance is involved . Once an access specifier has been used, it remains in effect until either another access specifier is encountered or the end of the class declaration is reached.
You may change access specifications as often as you like within a **class** declaration. For example, you may switch to **public** for some declarations and then switch back to **private** again. The class declaration in the following example illustrates this feature:

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

class employee
 {
char name[80]; // private by default
public:
void putname(char *n); // these are public
void getname(char *n);
private:
double wage; // now, private again
public:
void putwage(double w); // back to public
double getwage();
};
void employee::putname(char *n)
{
strcpy(name, n);
}
void employee::getname(char *n)
```

```
{
strcpy(n, name);
}
void employee::putwage(double w)
{
wage = w;
}
double employee::getwage()
{
return wage;
}

int main()
{
employee ted;
char name[80];
ted.putname("Ted Jones");
ted.putwage(75000);
ted.getname(name);
cout << name << " makes $";
cout << ted.getwage() << " per year.";
return 0;
}
```

Here, **employee** is a simple class that is used to store an employee's name and wage. Notice that the **public** access specifier is used twice.

Although you may use the access specifiers as often as you like within a class declaration, the only advantage of doing so is that by visually grouping various parts of a class, you may make it easier for someone else reading the program to understand it. However, to the compiler, using multiple access specifiers makes no difference. Actually, most programmers find it easier to have only one **private**, **protected**, and **public** section within each class. For example, most programmers would code the **employee** class as shown here, with all private elements grouped together and all public elements grouped together:

```
class employee
 {
char name[80];
double wage;
public:
void putname(char *n);
void getname(char *n);
void putwage(double w);
double getwage();
};
```

Functions that are declared within a class are **called *member functions***. Member functions may access any element of the class of which they are a part. This includes all **private** elements. Variables that are elements of a class are **called *member variables* or *data members*.** Collectively, any element of a class can be referred to as **a member** of that class.

There are a few restrictions that apply to class members. A non-**static** member variable cannot have an initializer. No member can be an object of the class that is being declared. (Although a member can be a pointer to the class that is being declared.) No member can be declared as **auto**, **extern**, or **register**.

In general, you should make all data members of a class private to that class. This is part of the way that encapsulation is achieved. However, there may be situations in which you will need to make one or more variables public. (For example, a heavily used variable may need to be accessible globally in order to achieve faster run times.) When a variable is public, it may be accessed directly by any other part of your program. The syntax for accessing a public data member is the same as for calling a member function: Specify the object's name, the dot operator, and the variable name.

This simple program illustrates the use of a public variable:

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

class myclass {
public:
int i, j, k; // accessible to entire program
};
int main()
{
myclass a, b;
a.i = 100; // access to i, j, and k is OK
a.j = 4;
a.k = a.i * a.j;
b.k = 12; // remember, a.k and b.k are different
cout << a.k << " " << b.k;
return 0;
}
```

Structures and Classes

Structures are part of the C subset and were inherited from the C language. As you have seen, a **class** is syntactically similar to a **struct**. But the relationship between a **class** and a **struct** is closer than you may at first think. In C++, the role of the structure was expanded, making it an alternative way to specify a class.

In fact, the only difference between a **class** and a **struct** is that by default all members are public in a**struct** and private in a **class**. In all other respects, structures and classes are equivalent.

The Scope Resolution Operator

As you know**, the :: operator** links a class name with a member name in order to tell the compiler what class the member belongs to. However, the scope resolution operator has another related use: it can allow access to a name in an enclosing scope that is "hidden" by a local declaration of the same name. For example, consider this fragment:

```
int i; // global i
void f()
{
int i; // local i
i = 10; // uses local i
.
.
.
}
```

As the comment suggests, the assignment **i = 10** refers to the local **i**. But what if function **f()** needs to access the global version of **i**? It may do so by preceding the **i** with the **::** operator, as shown here.

```
int i; // global i
void f()
{
int i; // local i
::i = 10; // now refers to global i
.
.
.
}
```

Friend Functions

It is possible to grant a non member function access to the private members of a class by using a **friend**. A **friend** function has access to all **private** and **protected** members of the class for which it is a **friend**. To declare a **friend** function, include its prototype within the class, preceding it with the keyword **friend**. Consider this program:

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

```
class myclass {
int a, b;
public:
friend int sum(myclass x);
void set_ab(int i, int j);
};

void myclass::set_ab(int i, int j)
{
a = i;
b = j;
}

// Note: sum() is not a member function of any class.
int sum(myclass x)
{

/* Because sum() is a friend of myclass, it can
directly access a and b. */

return x.a + x.b;
}
int main()
{
myclass n;
n.set_ab(3, 4);
cout << sum(n);
return 0;
}
```

In this example, the **sum()** function is not a member of **myclass**. However, it still has full access to its private members. Also, notice that **sum()** is called without the use of the dot operator. Because it is not a member function, it does not need to be (indeed, it may not be) qualified with an object's name.

Although there is nothing gained by making **sum()** a **friend** rather than amember function of **myclass**, there are some circumstances in which **friend** functions are quite valuable**. First**, friends can be useful when you are overloading certain types of operators. **Second**, **friend** functions make the creation of some types of I/O functions easier .

The third reason that **friend** functions may be desirable is that in some cases, two or more classes may contain members that are interrelated relative to other parts of your program.

Friend class

It is possible for one class to be a **friend** of another class. When this is the case, the **friend** class and all of its member functions have access to the private members defined within the other class. For example,

```
// Using a friend class.

#include <iostream>
using namespace std;

class TwoValues {
int a;
int b;
public:
TwoValues(int i, int j) { a = i; b = j; }
friend class Min;
};

class Min {
public:
int min(TwoValues x);
};

int Min::min(TwoValues x)
{
return x.a < x.b ? x.a : x.b;
}
int main()

{
TwoValues ob(10, 20);
Min m;
cout << m.min(ob);
return 0;
}
```

In this example, class **Min** has access to the private variables **a** and **b** declared within the **TwoValues** class.
It is critical to understand that when one class is a **friend** of another, it only has access to names defined within the other class. It does not inherit the other class. Specifically, the members of the first class do not become members of the **friend** class. Friend classes are seldom used. They are supported to allow certain special case situations to be handled.

Inline Functions

There is an important feature in C++, called **an *inline function***, that is commonly used with classes.

In C++, you can create short functions that are not actually called; rather, their code is expanded in line at the point of each invocation. This process is similar to using a function-like macro. To cause a function to be expanded in line rather than called, precede its definition with the **inline** keyword. For example, in this program, the function **max()** is expanded in line instead of called:

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

inline int max(int a, int b)
{
return a>b ? a : b;
}
int main()
{
cout << max(10, 20);
cout << " " << max(99, 88);
return 0;
}
```

As far as the compiler is concerned, the preceding program is equivalent to this one:

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

int main()
{
cout << (10>20 ? 10 : 20);
cout << " " << (99>88 ? 99 : 88);
return 0;
}
```

The reason that **inline** functions are an important addition to C++ is that they allow you to create very efficient code. Since classes typically require several frequently executed interface functions (which provide access to private data), the efficiency of these functions is of critical concern. As you probably know, each time a function is called, a significant amount of overhead is generated by the calling and return mechanism. Typically, arguments are pushed onto the stack and various registers are saved when a function is called, and then restored when the function returns. The trouble is that these instructions take time. However, when a function is expanded in line, none of those operations occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code. For this reason, it is best to **inline** only very small functions. Further, it is also a good idea to **inline** only those functions that will have significant impact on the

performance of your program.

Defining Inline Functions Within a Class

It is possible to define short functions completely within a class declaration. When a function is defined inside a class declaration, it is automatically made into an **inline** function (if possible). It is not necessary (but not an error) to precede its declaration with the **inline** keyword. For example, the preceding program is rewritten here with the definitions of **init()** and **show()** contained within the declaration of **myclass**:

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

class myclass {
int a, b;
public:
// automatic inline
void init(int i, int j) { a=i; b=j; }
void show() { cout << a << " " << b << "\n"; }
};

int main()
{
myclass x;
x.init(10, 20);
x.show();
return 0;
}
```

Notice the format of the function code within **myclass**. Because inline functions are short, this style of coding within a class is fairly typical. However, you are free to use any format you like.

Constructors and Destructors

It is very common for some part of an object to require initialization before it can be used. Because the requirement for initialization is so common, C++ allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor function.

A *constructor function* is a special function that is a member of a class and has the same name as that class. For example,

```
// This creates the class stack.

class stack {
int stck[SIZE];
int tos;
public:
stack(); // constructor
void push(int i);
int pop();
};
```

Notice that the constructor **stack()** has no return type specified. In C++, constructor functions cannot return values and, thus, have no return type.

The **stack()** function is coded like this:

```
// stack's constructor function
stack::stack()
{
tos = 0;
cout << "Stack Initialized\n";
}
```

Keep in mind that the message **Stack Initialized** is output as a way to illustrate the constructor. In actual practice, most constructor functions will not output or input anything. They will simply perform various initializations.

An object's constructor is automatically called when the object is created. This means that it is called when the object's declaration is executed. If you are accustomed to thinking of a declaration statement as being passive, this is not the case for C++. In C++, a declaration statement is a statement that is executed. This distinction is not just academic. The code executed to construct an object may be quite significant. An object's constructor is called once for global or **static** local objects. For local objects, the constructor is called each time the object declaration is encountered.

The complement of the constructor is the ***destructor***. In many circumstances, an object will need to perform some action or actions when it is destroyed. Local objects are created when their block is entered, and destroyed when the block is left. Global objects are destroyed when the program terminates. When an object is destroyed, its destructor (if it has one) is automatically called. There are many reasons why a destructor function may be needed. For example, an object may need to deallocate memory that it had previously allocated or it may need to close a file that it had opened. In C++, it is the destructor function that handles deactivation events. The

destructor has the same name as the constructor, but it is preceded by a ~. Forexample,

here is the **stack** class and its constructor and destructor functions. (Keep in mind that the **stack** class does not require a destructor; the one shown here is just for illustration.)

```
// This creates the class stack.

class stack {
int stck[SIZE];
int tos;
public:
stack(); // constructor
~stack(); // destructor
void push(int i);
int pop();
};
// stack's constructor function
stack::stack()
{
tos = 0;
cout << "Stack Initialized\n";
}
// stack's destructor function
stack::~stack()
{
cout << "Stack Destroyed\n";
}
```

Notice that, like constructor functions, destructor functions do not have return values.

Passing Objects to Functions

Objects may be passed to functions in just the same way that any other type of variable can. Objects are passed to functions through the use of the standard call-by value mechanism. This means that a copy of an object is made when it is passed to a function. However, the fact that a copy is created means, in essence, that another object is created. This raises the question of whether the object's constructor function is executed when the copy is made and whether the destructor function is executed when the copy is destroyed. The answer to these two questions may surprise you. To begin, here is an example:

```
// Passing an object to a function.
#include <iostream>
using namespace std;
```

```
class myclass {
int i;
public:
myclass(int n);
~myclass();
void set_i(int n) { i=n; }
int get_i() { return i; }
};
myclass::myclass(int n)
{
i = n;
cout << "Constructing " << i << "\n";
}
myclass::~myclass()
{
cout << "Destroying " << i << "\n";
}
void f(myclass ob);

int main()
{
myclass o(1);
f(o);
cout << "This is i in main: ";
cout << o.get_i() << "\n";
return 0;
}
void f(myclass ob)
{
ob.set_i(2);
cout << "This is local i: " << ob.get_i();
cout << "\n";
}
```

This program produces this output:

```
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying 1
```

Notice that two calls to the destructor function are executed, but only one call is made to the constructor function. As the output illustrates, the constructor function is not called when the copy of **o** (in **main()**) is passed to **ob** (within **f()**). The reason that the constructor function is not called when the copy of the object is made is easy to understand. When you pass an object to a function, you want the current state of that

object. If the constructor is called when the copy is created, initialization will occur, possibly changing the object. Thus, the constructor function cannot be executed when the copy of an object is generated in a function call.

Although the constructor function is not called when an object is passed to a function, it is necessary to call the destructor when the copy is destroyed. (The copy is destroyed like any other local variable, when the function terminates.) Remember, a new copy of the object has been created when the copy is made. This means that the copy could be performing operations that will require a destructor function to be called when the copy is destroyed.
For example, it is perfectly valid for the copy to allocate memory that must be freed when it is destroyed. For this reason, the destructor function must be executed when the copy is destroyed.

To summarize: When a copy of an object is generated because it is passed to a function, the object's constructor function is not called. However, when the copy of the object inside the function is destroyed, its destructor function is called.
By default, when a copy of an object is made, a bitwise copy occurs. This means that the new object is an exact duplicate of the original. The fact that an exact copy is made can, at times, be a source of trouble. Even though objects are passed to functions by means of the normal call-by-value parameter passing mechanism, which, in theory, protects and insulates the calling argument, it is still possible for a side effect to occur that may affect, or even damage, the object used as an argument. For example, if an object used as an argument allocates memory and frees that memory when it is destroyed, then its local copy inside the function will free the same memory when its destructor is called. This will leave the original object damaged and effectively useless. As explained later, it is possible to prevent this type of problem by defining the copy operation relative to your own classes by creating a special type of constructor called a ***copy constructor*.**

Returning Objects

A function may return an object to the caller. For example, this is a valid C++ program:

// Returning objects from a function.

```
#include <iostream>
using namespace std;
class myclass {
int i;
public:
void set_i(int n) { i=n; }
int get_i() { return i; }
};
myclass f(); // return object of type myclass
```

```
int main()
{
myclass o;
o = f();
cout << o.get_i() << "\n";
return 0;
}
myclass f()
{
myclass x;
x.set_i(1);
return x;
}
```

When an object is returned by a function, a temporary object is automatically created that holds the return value. It is this object that is actually returned by the function. After the value has been returned, this object is destroyed. The destruction of this temporary object may cause unexpected side effects in some situations. For example, if the object returned by the function has a destructor that frees dynamically allocated memory, that memory will be freed even though the object that is receiving the return value is still using it. There are ways to overcome this problem that involve overloading the assignment operator and defining a copy constructor.

Parameterized Constructors

It is possible to pass arguments to constructor functions. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object. For example, here is a simple class that includes a parameterized constructor:

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

class myclass {
int a, b;
public:
myclass(int i, int j) {a=i; b=j;}
void show() {cout << a << " " << b;}
};
int main()
{
myclass ob(3, 5);
ob.show();
return 0;
```

```
}
```

Notice that in the definition of **myclass(),** the parameters **i** and **j** are used to give initial values to **a** and **b**.

The program illustrates the most common way to specify arguments when you declare an object that uses a parameterized constructor function. Specifically, this statement
myclass ob(3, 4);
causes an object called **ob** to be created and passes the arguments **3** and **4** to the **i** and **j** parameters of **myclass().** You may also pass arguments using this type of declaration statement:
myclass ob = myclass(3, 4);

However, the first method is the one generally used, and this is the approach taken by most of the examples in this book.

Constructor with One Parameter

If a constructor only has one parameter, there is a third way to pass an initial value to that constructor. For example, consider the following short program.

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

class X {
int a;
public:
X(int j) { a = j; }
int geta() { return a; }
};
int main()
{
X ob = 99; // passes 99 to j
cout << ob.geta(); // outputs 99
return 0;
}
```

Here, the constructor for **X** takes one parameter. Pay special attention to how **ob** is declared in **main()** . In this form of initialization, 99 is automatically passed to the **j** parameter in the **X()** constructor. That is, the declaration statement is handled by the compiler as if it were written like this:
X ob = X(99);

In general, any time you have a constructor that requires only one argument, you can use either *ob*(*i*) or *ob* = *i* to initialize an object. The reason for this is that whenever you create a constructor that takes one argument, you are also implicitly creating a conversion from the type of that argument to the type of the class.
Remember that the alternative shown here applies only to constructors that have exactly one parameter.

Multiple Constructors

Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. Remember that for overloaded functions the compiler will call
the one whose parameters match the arguments used in the function call. In the case of constructors, which are automatically called when an object is created, the one executed is the one that matches the arguments passed on
the object declaration:

```
// overloading class constructors

#include <iostream>
using namespace std;

class CRectangle
{
int width, height;
public:
CRectangle ();
CRectangle (int,int);
int area (void)
 {return (width*height);}
};

CRectangle::CRectangle () {
width = 5;
height = 5;
}
CRectangle::CRectangle (int a, int b) {
width = a;
height = b;
}

int main ()
 {
```

```
CRectangle rect (3,4);
CRectangle rectb;
cout << "rect area: " << rect.area() << endl;
cout << "rectb area: " << rectb.area() << endl;
return 0;
}
```

```
rect area: 12
rectb area: 25
```

In this case, rectb was declared without any arguments, so it has been initialized with the constructor that has no parameters, which initializes both width and height with a value of 5.

Important: Notice how if we declare a new object and we want to use its default constructor (the one without parameters), we do not include parentheses ():

```
CRectangle rectb; // right
CRectangle rectb(); // wrong!
```

Copy constructor

One of the more important forms of an overloaded constructor is the *copy constructor*. Copy constructor is used to initialize and declare an object from another object.
For example,statement

```
Integer  i2(i1);
```

Would define the object i2 and athe same time iniyilize it to to the values of of i1.

Below is the simple example of constructing and using a copy constructor

```
#include<iostream.h>
Using namespace std;

Class code
{
Int id;
Public:
Code(){}                          // constructor
Code(int a){id=a;}                  //constructor again
Code(code&x)                       // copy constructor
{
Id=x.id;}
Void display()
```

```
{
Cout<<id
}
};
Int main()
{
Code a(100);              //object a is created and initialized
Code b(a);                //copy constructor called
Code c=a;                 //copy constructor called again
Code d;                   //d is created not initialized
D=a;                      // copy constructor not called

Cout<<"id of a";
a.display();
Cout<<"id of b";
b.display();
Cout<<"id of c";
c.display();
Cout<<"id of d";
d.display();

return 0;
}
```

Output:

```
Id of a :100
Id of b:100
Id of c:100
Id of d:100
```

Function Overloading

Function overloading is the process of using the same name for two or more functions. The secret to overloading is that each redefinition of the function must use either different types of parameters or a different number of parameters. It is only through these differences that the compiler knows which function to call in any given situation.

For example, this program overloads **myfunc()** by using different types of parameters.

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

```
int myfunc(int i); // these differ in types of parameters
double myfunc(double i);
int main()
{
cout << myfunc(10) << " "; // calls myfunc(int i)
cout << myfunc(5.4); // calls myfunc(double i)
return 0;
}
double myfunc(double i)
{
return i;
}
int myfunc(int i)
{
return i;
}
```

the key point about function overloading is that the functions must differ in regard to the types and/or number of parameters. Two functions differing only in their return types cannot be overloaded. For example, this is an invalid attempt to overload **myfunc():**

```
int myfunc(int i); // Error: differing return types are
float myfunc(int i); // insufficient when overloading.
```

Operator overloading

Closely related to function overloading is operator overloading. In C++, you can overload most operators so that they perform special operations relative to classes that you create. When an operator is overloaded, none of its original meanings are lost. Instead, the type of objects it can be applied to is expanded.

The ability to overload operators is one of C++'s most powerful features. It allows the full integration of new class types into the programming environment. After overloading the appropriate operators, you can use objects in expressions in just the same way that you use C++'s built-in data types. Operator overloading also forms the basis of C++'s approach to I/O.

You overload operators by creating operator functions. An ***operator function*** defines the operations that the overloaded operator will perform relative to the class upon which it will work. An operator function is created using the keyword **operator**. Operator functions can be either members or nonmembers of a class.

Nonmember operator functions are almost always friend functions of the class, however. The way operator functions are written differs between member and non member functions. Therefore, each will be examined separately, beginning with member operator functions.

Creating a Member Operator Function

A member operator function takes this general form:

*ret-type class-name::*operator*#(arg-list)*
{
// operations
}

Often, operator functions return an object of the class they operate on, but *ret-type* can be any valid type. The **#** is a placeholder. When you create an operator function, substitute the operator for the **#**. For example, if you are overloading the / operator, use **operator/.** When you are overloading a unary operator, *arg-list* will contain one parameter. (The reasons for this seemingly unusual situation will be made clear in a moment.)

Here is a simple first example of operator overloading. This program creates a class called **loc**, which stores longitude and latitude values. It overloads the **+** operator relative to this class. Examine this program carefully, paying special attention to the definition of **operator+()** :

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

class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
};
// Overload + for loc.
loc loc::operator+(loc op2)
{
loc temp;
```

```
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1.show(); // displays 10 20
ob2.show(); // displays 5 30
ob1 = ob1 + ob2;
ob1.show(); // displays 15 50
return 0;
}
```

As you can see, **operator+()** has only one parameter even though it overloads the binary **+** operator. (You might expect two parameters corresponding to the two operands of a binary operator.) The reason that **operator+()** takes only one parameter is that the operand on the left side of the **+** is passed implicitly to the function through the **this** pointer. The operand on the right is passed in the parameter **op2**. The fact that the left operand is passed using **this** also implies one important point: When binary operators are overloaded, it is the object on the left that generates the call to the operator function.

As mentioned, it is common for an overloaded operator function to return an object of the class it operates upon. By doing so, it allows the operator to be used in larger expressions. For example, if the **operator+()** function returned some other type, this expression would not have been valid:

```
ob1 = ob1 + ob2;
```

In order for the sum of **ob1** and **ob2** to be assigned to **ob1**, the outcome of that operation must be an object of type **loc**.

Further, having **operator+()** return an object of type **loc** makes possible the following statement:

```
(ob1+ob2).show(); // displays outcome of ob1+ob2
```

In this situation, **ob1+ob2** generates a temporary object that ceases to exist after the call to **show()** terminates.

It is important to understand that an operator function can return any type and that the type returned depends solely upon your specific application. It is just that, often, an operator function will return an object of the class upon which it operates.
One last point about the **operator+()** function: It does not modify either operand. Because the traditional use of the **+** operator does not modify either operand, it makes

sense for the overloaded version not to do so either. (For example, 5+7 yields 12, but neither 5 nor 7 is changed.) Although you are free to perform any operation you want inside an operator function, it is usually best to stay within the context of the normal use of the operator.

Overloading unary operator

Let us consider the unary minus operator.a minus operator when used as a unary takes just one operand.we will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an int or float variable.
The unary minus when applied to an object should change the sign of each of its data items.

```
#include<iostream.h>
Using namespace std;

 Class space {
Int x,y,z;
Public:
Void getdata(int a,int b,int c);
Void display();
Void operator-();          // overload unary minus
};

Void space::getdata(int a,int b,int c)
{
x=a;
y=b;
z=c;
}
Void space::display()
{
Cout<<x<<””;
Cout<<y<<””;
Cout<<z<<””;
}
Void space::operator-()
{
x=-x;
y=-y;
z=-z;
}

Int main()
{
```

```
Space s;
s.getdata(10,-20,30);
cout<<”s:”;
s.display();
-s;
Cout<<”s:”;
s.display();
return 0;
}
```
Out put:
S: 10 -20 30
S:-10 20 -30

Note: function operator-() takes no argument. then,what does this operator function do?it changes the sign of data members of the object s. since this function is a member function of the same class,it can directly access the members of the object which activated it.

Unit IV

Pointers to Objects

Just as you can have pointers to other types of variables, you can have pointers to objects. When accessing members of a class given a pointer to an object, use the arrow **(–>) operator** instead of the dot operator. The next program illustrates how to access an object given a pointer to it:

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


class cl
 {
int i;
```

```
public:
cl(int j) { i=j; }
int get_i() { return i; }
};

int main()
{
cl ob(88), *p;
p = &ob; // get address of ob
cout << p->get_i(); // use -> to call get_i()
return 0;
}
```

As you know, when a pointer is incremented, it points to the next element of its type. For example, an integer pointer will point to the next integer. In general, all pointer arithmetic is relative to the base type of the pointer. (That is, it is relative to the type of data that the pointer is declared as pointing to.) The same is true of pointers to objects.

The this Pointer

When a member function is called, it is automatically passed an implicit argument that is a pointer to the invoking object (that is, the object on which the function is called). This pointer is called **this**. The **this** pointer is very important when operators are overloaded and whenever a member function must utilize a pointer to the object that invoked it.

A complete program to illustrate the use of this is given in the program below:

```
   #include<iostream.h>
#include<cstring>
Using namespace std;

Class person
{
Char name[30];
Float age;

Public:
Person(char *s,float a)
{
Strcpy(name,s);
age=a;
}
Person &person:: greater (person &x)
{
If(x.age>=age)
```

```
return x;
Else
return *this;
}
Void display()
{
Cout<<”name:”<<name;
Cout<<”age:<<age;
}
};

Int main ()
{
Person p1(“john”,37.50),p2(“ahmed”,29),p3(“hebber”,40.25);

Person p=p1.greater(p3);    // p3.greater(p1)

Cout<<”elder person is”;
p.display();

p=p1.greater(p2);     // p2.greater(p1)
cout<<”elder person is”;
p.display();
return 0 ;
}
```

Out put:
Elder person is:
Name:hebber
Age:40.25
Elder person is:
Name: john
Age 37.5

Pointers to Derived class

In general, a pointer of one type cannot point to an object of a different type. However, there is an important exception to this rule that relates only to derived classes. To begin, assume two classes called **B** and **D**. Further, assume that **D** is derived from the base class **B**. In this situation, a pointer of type **B *** may also point to an object of type **D**. More generally, a base class pointer can also be used as a pointer to an object of any class derived from that base.

Although a base class pointer can be used to point to a derived object, the opposite is not true. A pointer of type **D *** may not point to an object of type **B**. Further, although

you can use a base pointer to point to a derived object, you can access only the members of the derived type that were imported from the base. That is, you won't be able to access any members added by the derived class. (You can cast a base pointer into a derived pointer and gain full access to the entire derived class, however.)

Here is a short program that illustrates the use of a base pointer to access derived objects.

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

class base {
int i;
public:
void set_i(int num) { i=num; }
int get_i() { return i; }
};

class derived: public base {
int j;
public:
void set_j(int num) { j=num; }
int get_j() { return j; }
};
int main()
{
base *bp;
derived d;

bp = &d;    // base pointer points to derived object
          // access derived object using base pointer

bp->set_i(10);

cout << bp->get_i() << " ";
    /* The following won't work. You can't access element of
    a derived class using a base class pointer.

bp->set_j(88);          // error
cout << bp->get_j();    // error
*/
return 0;
}
```

As you can see, a base pointer is used to access an object of a derived class.

polymorphism

Polymorphism is supported by C++ both at compile time and at run time.
compile-time polymorphism is achieved by overloading functions and operators. Run-time polymorphism is accomplished by using inheritance and virtual functions, and these are the topics of this chapter.

Polymorphism is one of the crucial features of oop.
It simply means one name multiple forms.we have already seen how the concept of polymorphism is implemented using the overloaded functions and operators.
The overloaded member functions are selected for invoking by matching arguments both type and number.this in information is known to the compiler at compile time and therefore compiler is able to select the appropriate function for a particular call at compile time itself. this is called **early binding or static binding or static linking**.also known **as compile time polymorphism**.where as in runtime polymorphism appropriate function is selected while program is running.this is known as **run time polymorphism.** .c++ supports a mechanism known as **virtual function** to achieve run time polymorphism.Since the function is linked with a particular class much later after the compilation this process is termed as **late binding**.it is laso known as **dynamic binding**.

Virtual Functions

A *virtual function* is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the *form* of the *interface* to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a *specific method*.
When accessed "normally," virtual functions behave just like any other type of class member function. However, what makes virtual functions important and capable of supporting run-time polymorphism is how they behave when accessed via a pointer.
As discussed earlier , a base-class pointer can be used to point to an object of any class derived from that base. When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon ***the type of object pointed to*** by the pointer. And this determination is made *at run time*. Thus, when different objects are pointed to, different versions of the virtual function are executed. The same effect applies to base-class references.

To begin, examine this short example:

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

class base {

public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {

public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};

class derived2 : public base {

public:
void vfunc() {
cout << "This is derived2's vfunc().\n";
}
};

int main()
{
base *p, b;
derived1 d1;
derived2 d2;

// point to base
p = &b;
p->vfunc(); // access base's vfunc()

// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // access derived2's vfunc()
return 0;
}
```

This program displays the following:

This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().

As the program illustrates, inside **base**, the virtual function **vfunc()** is declared. Notice that the keyword **virtual** precedes the rest of the function declaration. When **vfunc()** is redefined by **derived1** and **derived2**, the keyword **virtual** is not needed. (However, it is not an error to include it when redefining a virtual function inside a derived class; it's just not needed.)
In this program, **base** is inherited by both **derived1** and **derived2**. Inside each class definition, **vfunc()** is redefined relative to that class.
Inside **main(),** fourvariables are declared:

Name	**Type**
P	base class pointer
B	object of base
d1	object of derived1
d2	object of derived2

Next, **p** is assigned the address of **b,** and **vfunc()** is called via **p**. Since **p** is pointing to an object of type **base**, that version of **vfunc()** is executed. Next, **p** is set to the address of **d1**, and again **vfunc()** is called by using **p**. This time **p** points to an object of type **derived1**. This causes **derived1::vfunc()** to be executed. Finally, **p** is assigned the address of **d2,** and **p−>vfunc()** causes the version of **vfunc()** redefined inside **derived2** to be executed. The key point here is that the kind of object to which **p** points determines which version of **vfunc()** is executed. Further, this determination is made at run time, and this process forms the basis for run-time polymorphism.
Although you can call a virtual function in the "normal" manner by using an object's name and the dot operator, it is only when access is through a base-class pointer (or reference) that run-time polymorphism is achieved. For example, assuming the preceding example, this is syntactically valid:

```
d2.vfunc(); // calls derived2's vfunc()
```

Although calling a virtual function in this manner is not wrong, it simply does not take advantage of the virtual nature of **vfunc()** .

At first glance, the redefinition of a virtual function by a derived class appears similar to function overloading. However, this is not the case, and the term *overloading* is not applied to virtual function redefinition because several differences exist. Perhaps the most important is that the prototype for a redefined virtual function must match exactly the prototype specified in the base class. This differs from overloading a normal function, in which return types and the number and type of parameters may differ. (In fact, when you overload a function, either the number or the type of the parameters *must* differ! It is through these differences that C++ can select the correct version of an overloaded

function.) However, when a virtual function is redefined, all aspects of its prototype must be the same. If you change the prototype when you attempt to redefine a virtual function, the function will simply be considered overloaded by the C++ compiler, and its virtual nature will be lost. Another important restriction is that virtual functions must be nonstatic members of the classes of which they are part. They cannot be **friend**s. Finally,constructor functions cannot be virtual, but destructor functions can.
Because of the restrictions and differences between function overloading and virtual function redefinition, the term *overriding* is used to describe virtual function redefinition by a derived class.

Pure Virtual Functions

As the examples in the preceding section illustrate, when a virtual function is not redefined by a derived class, the version defined in the base class will be used. However, in many situations there can be no meaningful definition of a virtual function within a base class. For example, a base class may not be able to define an object sufficiently to allow a base-class virtual function to be created. Further, in some situations you will want to ensure that all derived classes override a virtual function. To handle these two cases, C++ supports the pure virtual function.

A ***pure virtual function*** is a virtual function that has no definition within the base class. To declare a pure virtual function, use this general form:

virtual *type func-name(parameter-list*) = 0;

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result.

Abstract Classes

A class that contains at least one pure virtual function is said to be *abstract*. Because an abstract class contains one or more functions for which there is no definition (that is, a pure virtual function), no objects of an abstract class may be created. Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes.
Although you cannot create objects of an abstract class, you can create pointers and references to an abstract class. This allows abstract classes to support run-time polymorphism, which relies upon base-class pointers and references to select the proper virtual function.

Early vs. Late Binding

Before concluding this chapter on virtual functions and run-time polymorphism, there are two terms that need to be defined because they are used frequently in discussions of C++ and object-oriented programming: *early binding* and *late binding*.

Early binding refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is known at compile time. (Put differently, early binding means that an object and a function call are bound during compilation.) Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators. The main advantage to early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast. The opposite of early binding is *late binding*. As it relates to C++, late binding refers to function calls that are not resolved until run time. Virtual functions are used to achieve late binding. As you know, when access is via a base pointer or reference, the virtual function actually called is determined by the type of object pointed to by the pointer. Because in most cases this cannot be determined at compile time, the object and the function are not linked until run time. The main advantage to late binding is flexibility. Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code." Keep in mind that because a function call is not resolved until run time, late binding can make for somewhat slower execution times.

Inheritance

Inheritance is one of the cornerstones of OOP because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class may then be inherited by other, more specific classes, each adding only those things that are unique to the inheriting class.
In keeping with standard C++ terminology, a class that is inherited is referred to as a ***base class*.** The class that does the inheriting is called the ***derived class***. Further, a derived class can be used as a base class for another derived class. In this way, multiple inheritance is achieved.
C++'s support of inheritance is both rich and flexible.

Base-Class Access Control

When a class inherits another, the members of the base class become members of the derived class.
Class inheritance uses this general form:

class *derived-class-name : access base-class-name* {
// *body of class*
};

The access status of the base-class members inside the derived class is determined by *access*. The base-class access specifier must be either **public, private,** or **protected**. If no access specifier is present, the access specifier is **private** by default if the derived class is a **class**. If the derived class is a **struct**, then **public** is the default in the absence of an explicit access specifier. Let's examine the ramifications of using **public** or **private** access. (The **protected** specifier is examined in the next section.)

When the access specifier for a base class is **public**, all public members of the base become public members of the derived class, and all protected members of the base become protected members of the derived class. In all cases, the base's private elements remain private to the base and are not accessible by members of the derived class.

For example, as illustrated in this program, objects of type **derived** can directly access the public members of **base**(single inheritance)

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

class base
 {
int i, j;

public:
void set(int a, int b) { i=a; j=b; }

void show() { cout << i << " " << j << "\n"; }
};

class derived : public base {

int k;
public:
derived(int x) { k=x; }
void showk() { cout << k << "\n"; }
};

int main()
{
derived ob(3);
ob.set(1, 2); // access member of base
ob.show(); // access member of base
ob.showk(); // uses member of derived class
return 0;
}
```

When the base class is inherited by using the **private** access specifier, all public and protected members of the base class become private members of the derived class.

Note: ***When a base class' access specifier is private, public and protected members of the base become private members of the derived class. This means that they are still accessible by members of the derived class but cannot be accessed by parts of your program that are not members of either the base or derived class.***

Inheritance and protected Members

The **protected** keyword is included in C++ to provide greater flexibility in the inheritance mechanism. When a member of a class is declared as **protected**, that member is not accessible by other, nonmember elements of the program. With one important exception, access to a protected member is the same as access to a private member—it can be accessed only by other members of its class. The sole exception to this is when a protected member is inherited. In this case, a protected member differs substantially from a private one.
As explained in the preceding section, a private member of a base class is not accessible by other parts of your program, including any derived class. However, protected members behave differently. If the base class is inherited as **public**, then the base class' protected members become protected members of the derived class and are, therefore, accessible by the derived class. By using **protected**, you can create class members that are private to their class but that can still be inherited and accessed by a derived class.

When a derived class is used as a base class for another derived class, any protected member of the initial base class that is inherited (as public) by the first derived class may also be inherited as protected again by a second derived class. For example,
derived2 does indeed have access to **i** and **j**.

```
#include <iostream>       // example of multilevel inheritance
using namespace std;

class base {

protected:
int i, j;

public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};

// i and j inherited as protected.
class derived1 : public base
{
```

```
int k;
public:
void setk() { k = i*j; } // legal
void showk() { cout << k << "\n"; }
};
// i and j inherited indirectly through derived1.
class derived2 : public derived1
 {
int m;
public:
void setm() { m = i-j; } // legal
void showm() { cout << m << "\n"; }
};

int main()
{
derived1 ob1;
derived2 ob2;
ob1.set(2, 3);
ob1.show();
ob1.setk();
ob1.showk();
ob2.set(3, 4);
ob2.show();
ob2.setk();
ob2.setm();
ob2.showk();
ob2.showm();
return 0;
}
```

If, however, **base** were inherited as **private**, then all members of **base** would become private members of **derived1**, which means that they would not be accessible by **derived2**. (However, **i** and **j** would still be accessible by **derived1**.)

Multiple Inheritance

It is possible for a derived class to inherit two or more base classes. For example, in this short example, **derived** inherits both **base1** and **base2**.

```
// An example of multiple base classes.

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

```
class base1
{
protected:
int x;
public:
void showx()
{ cout << x << "\n"; }
};

class base2
{
protected:
int y;

public:
void showy() {cout << y << "\n";}
};

// Inherit multiple base classes.

class derived: public base1, public base2
 {
public:
void set(int i, int j) { x=i; y=j; }
};

int main()
{
derived ob;
ob.set(10, 20); // provided by derived
ob.showx(); // from base1
ob.showy(); // from base2
return 0;
}
```

As the example illustrates, to inherit more than one base class, use a comma-separated list. Further, be sure to use an access-specifier for each base inherited.

Virtual Base Classes

An element of ambiguity can be introduced into a C++ program when multiple base classes are inherited.

When two or more objects are derived from a common base class, we can prevent multiple copies of the base class from being present in an object derived from those

objects by declaring the base class as **virtual** when it is inherited. we accomplish this by preceding the base class' name with the keyword **virtual** when it is inherited.
The duplication of inherited members due to the multiple paths can be avoided by making the common base class(ancestor class) as **virtual base class**. For example, here is the program in which **derived3** contains only one copy of **base:**

```
// This program uses virtual base classes.

#include <iostream>
using namespace std;

class base
{
public:
int i;
};

// derived1 inherits base as virtual.
class derived1 : virtual public base
{
public:
int j;
};

// derived2 inherits base as virtual.
class derived2 : virtual public base
 {
public:
int k;
};

/* derived3 inherits both derived1 and derived2.
This time, there is only one copy of base class. */
class derived3 : public derived1, public derived2
 {
public:
int sum;
};

int main()
{
derived3 ob;
ob.i = 10; // now unambiguous
ob.j = 20;
ob.k = 30;
```

```
// unambiguous
ob.sum = ob.i + ob.j + ob.k;

// unambiguous
cout << ob.i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

As you can see, the keyword **virtual** precedes the rest of the inherited **class**' specification. Now that both **derived1** and **derived2** have inherited **base** as **virtual**, any multiple inheritance involving them will cause only one copy of **base** to be present. Therefore, in **derived3**, there is only one copy of **base** and **ob.i = 10** is perfectly valid and unambiguous.

One further point to keep in mind: Even though both **derived1** and **derived2** specify **base** as **virtual**, **base** is still present in objects of either type