Table of Content

Lesson 1

Object oriented programming

The prime purpose of C++ programming was to add object orientation to the C programming language, which is in it one of the most powerful programming languages.

The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods. While designing C++ modules, we try to see whole world in the form of objects. For example a car is an object which has certain properties such as color, number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

There are few principle concepts that form the foundation of object-oriented programming:

Object:

This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.

Class:

When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

Abstraction:

Data abstraction refers to, providing only essential information to the outside word and hiding their background details ie. to represent the needed information in program without presenting the details.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

Encapsulation:

Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

Inheritance:

One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.

This is a very important concept of object oriented programming since this feature helps to reduce the code size.

Polymorphism:

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

Overloading:

The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type it is said to be overloaded.

Introduction of C++

C++ is a statically typed, compiled, general purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming.

C++ is regarded as a **middle-level** language, as it comprises a combination of both high-level and low-level language features.

C++ was developed by Bjarne Stroustrup started in 1979 at Bell Labs in Murray Hill, New Jersey as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.

C++ is a superset of C, and that virtually any legal C program is a legal C++ program.

Note: A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time.

Object-Oriented Programming:

C++ fully supports object-oriented programming, including the four pillars of object-oriented development:

1. Encapsulation
2. Data hiding
3. Inheritance
4. Polymorphism

Standard Libraries:

Standard C++ consists of three important parts:

1. The core language giving all the building blocks including variables, data types and literals etc.
2. The C++ Standard Library giving a rich set of functions manipulating files, strings etc.
3. The Standard Template Library (STL) giving a rich set of methods manipulating data structures etc.

The ANSI Standard:

The ANSI standard is an attempt to ensure that C++ is portable -- that code you write for Microsoft's compiler will compile without errors, using a compiler on a Mac, Unix, a Windows box, or an Alpha.

The ANSI standard has been stable for a while, and all the major C++ compiler manufacturers support the ANSI standard.

Learning C++:

The most important thing to do when learning C++ is to focus on concepts and not get lost in language technical details.

The purpose of learning a programming language is to become a better programmer; that is, to become more effective at designing and implementing new systems and at maintaining old ones.

C++ supports a variety of programming styles. You can write in the style of Fortran, C, Smalltalk, etc., in any language. Each style can achieve its aims effectively while maintaining runtime and space efficiency.

Use of C++:

C++ is used by hundreds of thousands of programmers in essentially every application domain.

C++ is being highly used to write device drivers and other software that rely on direct manipulation of hardware under real time constraints.

C++ is widely used for teaching and research because it is clean enough for successful teaching of basic concepts.

Anyone who has used either an Apple Macintosh or a PC running Windows has indirectly used C++ because the primary user interfaces of these systems are written in C++.

Structure of a program

Probably the best way to start learning a programming language is by writing a program. Therefore, here is our
first program:

```
// my first program in C++
#include <iostream>
using namespace std;
int main ()
{
cout << "Hello World!";
return 0;
}
```

Hello World!

The first panel shows the source code for our first program. The second one shows the result of the program once compiled and executed. The way to edit and compile a program depends on the compiler you are using. Depending on whether it has a

Development Interface or not and on its version. Consult the compilers section and the manual or help included with your compiler if you have doubts on how to compile a C++ console program. The previous program is the typical program that programmer apprentices write for the first time, and its result is the printing on screen of the "Hello World!" sentence. It is one of the simplest programs that can be written in C++, but it already contains the fundamental components that every C++ program has. We are going to look line by line at the code we have just written:

// my first program in C++

This is a comment line. All lines beginning with two slash signs (//) are considered comments and do not have any effect on the behavior of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

#include <iostream.h>

Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. In this case the directive #include <iostream.h> tells the preprocessor to include the iostream standard file. This specific file (iostream.h) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

int main ()

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it – the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a main function. The word main is followed in the code by a pair of parentheses (()). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of expressions are these parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them.

Right after these parentheses we can find the body of the main function enclosed in braces ({}). What is contained within these braces is what the function does when it is executed.

cout << "Hello World!";

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program.

cout represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (which usually is the screen).

cout is declared in the iostream standard file within the std namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code.

Notice that the statement ends with a semicolon character (;). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs (one of the most common syntax errors is indeed to forget to include some semicolon after a statement).

return 0;

The return statement causes the main function to finish. return may be followed by a return code (in our example is followed by the return code 0). A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

You may have noticed that not all the lines of this program perform actions when the code is executed. There were lines containing only comments (those beginning by //). There were lines with directives for the compiler's preprocessor (those beginning by #). Then there were lines that began the declaration of a function (in this case, the main function) and, finally lines with statements (like the insertion into cout), which were all included within the block delimited by the braces ({}) of the main function.
The program has been structured in different lines in order to be more readable, but in C++, we do not have strict rules on how to separate instructions in different lines. For example, instead of

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

We could have written:

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

All in just one line and this would have had exactly the same meaning as the previous code.
In C++, the separation between statements is specified with an ending semicolon (;) at the end of each one, so the separation in different code lines does not matter at all for this purpose. We can write many statements per line or write a single statement that takes many code lines. The division of code in different lines serves only to make it more legible and schematic for the humans that may read it. Let us add an additional instruction to our first program:

```
// my second program in C++
#include <iostream>
using namespace std;
int main ()
{
cout << "Hello World! ";
cout << "I'm a C++ program";
return 0;
}
```

Hello World! I'm a C++ program
In this case, we performed two insertions into cout in two different statements. Once again, the separation in different lines of code has been done just to give greater readability to the program, since main could have been perfectly valid defined this way:

```
int main ()
{
 cout << " Hello World! ";
 cout << " I'm a C++ program ";
 return 0;
}
```

And the result would again have been exactly the same as in the previous examples.

Preprocessor directives (those that begin by #) are out of this general rule since they are not statements. They are lines read and processed by the preprocessor and do not produce any code by themselves. Preprocessor directives must be specified in their own line and do not have to end with a semicolon (;).

Comments

Comments are parts of the source code disregarded by the compiler. They simply do nothing. Their purpose is only
to allow the programmer to insert notes or descriptions embedded within the source code.
C++ supports two ways to insert comments:

// line comment
/* block comment */

The first of them, known as line comment, discards everything from where the pair of slash signs (//) is found up
to the end of that same line. The second one, known as block comment, discards everything between the /*
Characters and the first appearance of the */ characters, with the possibility of including more than one line.
We are going to add comments to our second program:

```
/* my second program in C++
with more comments */
#include <iostream>
using namespace std;
int main ()
{
cout << "Hello World! "; // prints Hello
World!
cout << "I'm a C++ program"; // prints I'm a
C++ program
return 0;
}
```

Hello World! I'm a C++ program

If you include comments within the source code of your programs without using the comment characters combinations //, /* or */, the compiler will take them as if they were C++ expressions, most likely causing one or several error messages when you compile it.

Lesson 2

Variables and Data Types in C++ programming

Variables

"Variable is a memory location in C++ Programming language" Variable are use to store data on memory on which user can perform operation to need.
Variables are use to store value and that values can be change. The values of variables can be numeric or alphabets.
There are certain rules on choosing variable name:
=> Variable name can consist of letter, alphabets and underscore character.
=> First character of variable should always be alphabet and cannot be digit.
=> Blank spaces are not allowed in variable name.
=> Special characters like #, $ are not allowed.
=> A single variable can only be declare for only 1 data type in a program.
=> As C++ is case sensitive language so if we declare a variable name and one more NAME both are two different variables.
=> C++ has certain key words which cannot be use for variable name.
=> A variable name can be consist of 31 characters only if we declare a variable more than 1 characters compiler will ignore after 31 characters.
Recommendation in Variable naming:
Always declare variable in readable form e.g. name ,sale etc.

Data type in C++ language

"A data type defines which kind of data will store in variables and also defines memory storage of data"
There are two kind of data types User defined data types and Standard data types. In the beginning we will discuss only standard data types only.

In C++ language there are four main data types

1.int
2.float
3.double
4.char

Character data type

A keyword char is used for character data type. This data type is used to represents letters or symbols. A character variable occupies 1 byte of memory.

Sensitive case:-
If we chose char data type then we try to store an integer in it like 5.then compiler will given an error if we put single quotes around 5 like that '5'.then compiler will store it as an character not integer.
There is an ASCII value for every character in C++ Language. Characters are store on these ASCII values in memory. On the basis of these ASCII values characters can be compare,

subtract or add.

Integer data types

Integers are those values which has no decimal par they can be positive or negative. Like 12or -12.

There are 3 data types for integers

1.int
2.short int
3.long int

int

=>int keyword is used for integers.
=>It takes two bytes in memory.

there are two more types of int data type

a)Signed int or short int(2nd type of integer data type)
b)Unsigned int or unsigned short int

Signed int

The range of storing value of signed int variable is -32768 to 32767.It can interact with both positive and negative value.

Unsigned int

This type of integers cannot handle negative values.its rang is 0 to 65535.

Long int

=>As name refers it is used to represent larger integers.

=>It takes 4byte in memory.

=>The range of long int is -2147483648 to 2147483648.

Float Data type

=>this type of integers contain fractional part.

=>There are two further type of float integer's number.

Below table showing:
=> Number of bytes or memory taken by numbers
=> Decimal places up to which they can represent value
=> Range

Data type	Bytes	Decimal places	Range of values
float	4	6	3.4E -38 to3.4E+38
double	8	15	1.7E - 308 to1.7E +308
long double	10	19	3.4E -4932 to 304E +4932

Lesson 3

Operators

Operators

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:

```
1 // assignment operator
2
3 #include <iostream.h>
4
```

```
a:4 b:7
```

```
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;
}
```

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-left rule).

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 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.h>

int main ()

{

  int a, b=3;

  a = b;

  a+=2;            // equivalent to a=a+2

  cout << a;

  return 0;

}
```

Increment and decrement (++, --)

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

```
1 c++;
2 c+=1;
3 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;	B=3; A=B++;

`// A contains 4, B contains 4`	`// 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 and then 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 an assignment 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 following panel shows the result of operator && evaluating the expression a && b:

&& OPERATOR

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

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).
```

When using the logical operators, C++ only evaluates what is necessary from left to right to come up with the combined relational result, ignoring the rest. Therefore, in this last example ((5==5)||(3>6)), C++ would evaluate first whether 5==5 is true, and if so, it would never check whether 3>6 is true or not. This is known as short-circuit evaluation, and works like this for these operators:

operator	short-circuit
&&	if the left-hand side expression is false, the combined result is false (right-hand side expression not evaluated).
||	if the left-hand side expression is true, the combined result is true (right-hand side expression not evaluated).

This is mostly important when the right-hand expression has side effects, such as altering values:

```
if ((i<10)&&(++i<n)) { /*...*/ }
```


This combined conditional expression increases i by one, but only if the condition on the left of && is true, since otherwise the right-hand expression (++i<n) is never evaluated.

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;
}
```

```
7
```

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 expression is 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 (()):

```
1 int i;
2 float f = 3.14;
3 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.

Other operators

Later in these tutorials, we will see a few more operators, like the ones referring to pointers or the specifics for object-oriented programming. Each one is treated in its respective section.

Precedence of operators

When writing complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. For example, in this expression:

```
a = 5 + 7 % 2
```

we may doubt if it really means:

```
a = 5 + (7 % 2)     // with a result of 6, or
a = (5 + 7) % 2     // with a result of 0
```

The correct answer is the first of the two expressions, with a result of 6. There is an established order with the priority of each operator, and not only the arithmetic ones (those whose preference come from mathematics) but for all the operators which can appear in C++. From greatest to lowest priority, the priority order is as follows:

Level	Operator	Description	Grouping
1	::	scope	Left-to-right
2	() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	postfix	Left-to-right
3	++ -- ~ ! sizeof new delete	unary (prefix)	Right-to-left
	* &	indirection and reference (pointers)	
	+ -	unary sign operator	
4	(type)	type casting	Right-to-left
5	.* ->*	pointer-to-member	Left-to-right
6	* / %	multiplicative	Left-to-right
7	+ -	additive	Left-to-right
8	<< >>	shift	Left-to-right

9	< > <= >=	relational	Left-to-right
10	== !=	equality	Left-to-right
11	&	bitwise AND	Left-to-right
12	^	bitwise XOR	Left-to-right
13	|	bitwise OR	Left-to-right
14	&&	logical AND	Left-to-right
15	||	logical OR	Left-to-right
16	?:	conditional	Right-to-left
17	= *= /= %= += -= >>= <<= &= ^= |=	assignment	Right-to-left
18	,	comma	Left-to-right

Grouping defines the precedence order in which operators are evaluated in the case that there are several operators of the same level in an expression.

All these precedence levels for operators can be manipulated or become more legible by removing possible ambiguities using parentheses signs (and), as in this example:


```
a = 5 + 7 % 2;
```

might be written either as:

```
a = 5 + (7 % 2);
```

or

```
a = (5 + 7) % 2;
```

Lesson 4

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.h>
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): T=36

1. User assigns a value to n

2. The while condition is checked (n>0). At this point there are two possibilities:

* 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.h>

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; condition; increment/Decrement)

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.h>

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.

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.h>
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.h>

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.h>

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 exit function

exit is a function defined in the c stdlib library.

The purpose of exit is to terminate the current program with a specific exit code. Its prototype is:

void exit (int exitcode);

The exitcode is used by some operating systems and may be used by calling programs. By convention, an exit code of 0 means that the program finished normally and any other value means that some error or unexpected results happened.

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.

If you need to check ranges or values that are not constants, use a concatenation of if and else if statements.

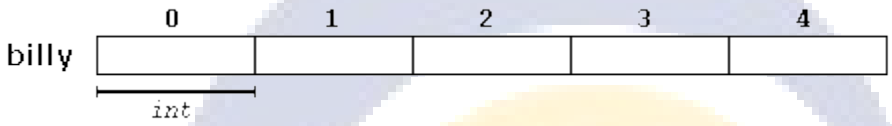
Lesson 5

Arrays

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

That means that, for example, we can store 5 values of type `int` in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, using an array we can store 5 different values of the same type, `int` for example, with a unique identifier.

For example, an array to contain 5 integer values of type `int` called `billy` could be represented like this:



where each blank panel represents an element of the array, that in this case are integer values of type `int`. These elements are numbered from `0` to `4` since in arrays the first index is always `0`, independently of its length.

Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

```
type name [elements];
```

where `type` is a valid type (like `int`, `float`...), `name` is a valid identifier and the `elements` field (which is always enclosed in square brackets `[]`), specifies how many of these elements the array has to contain.

Therefore, in order to declare an array called `billy` as the one shown in the above diagram it is as simple as:

```
int billy [5];
```

NOTE: The `elements` field within brackets `[]` which represents the number of elements the array is going to hold, must be a **constant** value, since arrays are blocks of non-dynamic memory whose size must be determined before execution. In order to create arrays with a variable length dynamic memory is needed, which is explained later in these tutorials.

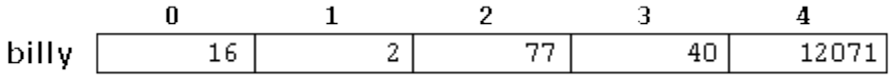
Initializing arrays.

When declaring a regular array of local scope (within a function, for example), if we do not specify otherwise, its elements will not be initialized to any value by default, so their content will be undetermined until we store some value in them. The elements of global and static arrays, on the other hand, are automatically initialized with their default values, which for all fundamental types this means they are filled with zeros.

In both cases, local and global, when we declare an array, we have the possibility to assign initial values to each one of its elements by enclosing the values in braces `{ }`. For example:

```
int billy [5] = { 16, 2, 77, 40, 12071 };
```

This declaration would have created an array like this:



The amount of values between braces `{ }` must not be larger than the number of elements that we declare for the array between square brackets `[ ]`. For example, in the example of array `billy` we have declared that it has 5 elements and in the list of initial values within braces `{ }` we have specified 5 values, one for each element.

When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty `[ ]`. In this case, the compiler will assume a size for the array that matches the number of values included between braces `{ }`:

```
int billy [] = { 16, 2, 77, 40, 12071 };
```

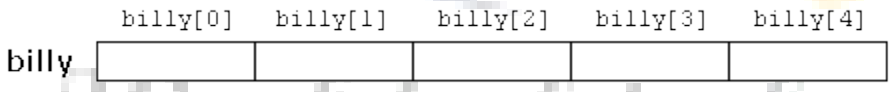
After this declaration, array `billy` would be 5 ints long, since we have provided 5 initialization values.

Accessing the values of an array.

In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value. The format is as simple as:

`name[index]`

Following the previous examples in which `billy` had 5 elements and each of those elements was of type `int`, the name which we can use to refer to each element is the following:



For example, to store the value `75` in the third element of `billy`, we could write the following statement:

```
billy[2] = 75;
```

and, for example, to pass the value of the third element of `billy` to a variable called `a`, we could write:

```
a = billy[2];
```

Therefore, the expression `billy[2]` is for all purposes like a variable of type `int`.

Notice that the third element of `billy` is specified `billy[2]`, since the first one is `billy[0]`, the second one

is`billy[1]`, and therefore, the third one is `billy[2]`. By this same reason, its last element is `billy[4]`. Therefore, if we write billy[5], we would be accessing the sixth element of `billy` and therefore exceeding the size of the array.

In C++ it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause compilation errors but can cause runtime errors. The reason why this is allowed will be seen further ahead when we begin to use pointers.

At this point it is important to be able to clearly distinguish between the two uses that brackets `[ ]` have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements. Do not confuse these two possible uses of brackets `[ ]` with arrays.

```
int billy[5];          // declaration of a new array
billy[2] = 75;         // access to an element of the array.
```

If you read carefully, you will see that a type specifier always precedes a variable or array declaration, while it never precedes an access.

Some other valid operations with arrays:

```
billy[0] = a;
billy[a] = 75;
b = billy [a+2];
billy[billy[a]] = billy[2] + 5;
```

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

int billy [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; n++ )
  {
    result += billy[n];
  }
  cout << result;
  return 0;
}
```

```
12206
```

Types of Array

1.One-Dimentional Array

A one-D array is having a single subscript only.

In above example num array is declared having 5 elements, the first element of array can be represented as num[0]. His can be displayed as:

num

12	15	77	80	70
0	1	2	3	4

```
#include<iostream.h>
#include<conio.h>
void main()
{
float num[5];
int i,j;
cout<<"enter the number of array";
for(i=0;i<5;i++)
{
cin>>num[i];
}
cout<<"displaying values in reverse order";
for(i=4;i>=0;i--)
{
cout<<"\t"<<num[i];
cout<<endl;
}
getch();
```

```
}
```

Two-Dimensional Array

In double dimensional array the first index represents the rows and second index represents column.

```
int num[3][10];
```

eg.

```
#include<iostream.h>
#include<conio.h>
void main()
{
int sales[3][3];
int i,j,total;
for(i=0;i<3;i++)
{
total=0;
cout<<"enter sales for salesman"<<i+1<<endl;
for(j=0;j<3;j++)
{
cout<<"month"<<j+1 <<endl;
cin>>sales[i][j];
total=total+sales[i][j];
}
cout<<"total sales of salesman"<<i+1<<"="<<total<<endl;
}
getch();
}
```

Output:

```
enter sales for salesman1
month1
200
month2
300
month3
400
total sales of salesman1=900
enter sales for salesman2
month1
```

Ques: Write a program to perform matrix addition using 2-d array.

```
#include<iostream.h>
#include<conio.h>
void main()
{
int A[2][2], B[2][2], C[2][2], i, j, m, n;
clrscr();
cout<<"\n\n\t ENTER A ORDER OF THE MATRICES M,N…:";
cin>>m>>n;
cout<<"\n\n\t ENTER THE ELEMENTS OF THE FIRST MATRIX..:\n\n";
for(i=1;i<=m;i++)
{
for(j=1;j<=n;j++)
{
cin>>A[i][j];
}
cout<<endl;
}
cout<<"\n\n\t ENTER THE ELEMENTS OF THE second MATRIX..:\n\n";
for(i=1;i<=m;i++)
{
for(j=1;j<=n;j++)
```

```
{


cin>>B[i][j];
}
cout<<endl;
}
for(i=1;i<=m;i++)
{
for(j=1;j<=n;j++)
{
C[i][j] = A[i][j] + B[i][j];
}
}
cout<<"\n\t THE SUM OF TWO MATRICES IS…:\n\n\t\t\t ";
for(i=1;i<=m;i++)
{
for(j=1;j<=n;j++){
cout<<C[i][j];}
cout<<"\n\n\t\t\t ";}
getch();
}
```

Lesson 6

Functions

Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming can offer to us in C++.

A function is a group of statements that is executed when it is called from some point of the program.

The following is its format:

Type name (parameter1, parameter2, ...) { statements }

Where:

• Type is the data type specifier of the data returned by the function.

• Name is the identifier by which it will be possible to call the function.

• Parameters (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: int x) and which acts within the function as a regular local variable. They allow passing arguments to the function when it is called. The different parameters are separated by commas.

• Statements is the function's body. It is a block of statements surrounded by braces { }.

Here you have the first function example:

```
// function example
#include <iostream.h>
int addition (int a, int b)
{
  int r;
  r=a+b;
return (r);
}
int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
```

```
  return 0;

}
```

The result is 8

In order to examine this code, first of all remember something said at the beginning of this tutorial: a C++ program always begins its execution by the main function. So we will begin there.

We can see how the main function begins by declaring the variable z of type int. Right after that, we see a call to a function called addition. Paying attention we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself some code lines above:

The parameters and arguments have a clear correspondence. Within the main function we called to addition passing two values: 5 and 3, that correspond to the int a and int b parameters declared for function addition.

At the point at which the function is called from within main, the control is lost by main and passed to function addition. The value of both arguments passed in the call (5 and 3) are copied to the local variables int a and int b within the function.

Function addition declares another local variable (int r), and by means of the expression r=a+b, it assigns to r the result of a plus b. Because the actual parameters passed for a and b are 5 and 3 respectively, the result is 8.

The following line of code:

return (r);

Finalizes function addition, and returns the control back to the function that called it in the first place (in this case, main). At this moment the program follows it regular course from the same point at which it was interrupted by the call to addition. But additionally, because the return statement in function addition specified a value: the content of variable r (return (r);), which at that moment had a value of 8. This value becomes the value of evaluating the function call.

So being the value returned by a function the value given to the function call itself when it is evaluated, the variable z will be set to the value returned by addition (5, 3), that is 8. To explain it another way, you can imagine that the call to a function (addition (5,3)) is literally replaced by the value it returns (8).

The following line of code in main is:

cout << "The result is " << z;

That, as you may already expect, produces the printing of the result on the screen.

Scope of variables

The scope of variables declared within a function or any other inner block is only their own function or their own block and cannot be used outside of them. For example, in the previous example it would have been impossible to use the variables a, b or r directly in function main since they were variables local to function addition. Also, it would have been

impossible to use the variable z directly within function addition, since this was a variable local to the function main.

Therefore, the scope of local variables is limited to the same block level in which they are declared. Nevertheless, we also have the possibility to declare global variables; These are visible from any point of the code, inside and outside all functions. In order to declare global variables you simply have to declare the variable outside any function or block; that means, directly in the body of the program.

And here is another example about functions:

```
// function example
#include <iostream.h>
int subtraction (int a, int b)
{
  int r;
  r=a-b;
return (r);
}
int main ()
{
  int x=5, y=3, z;
  z = subtraction (7,2);
  cout << "The first result is " << z << '\n';
  cout << "The second result is " << subtraction (7,2) << '\n';
  cout << "The third result is " << subtraction (x,y) << '\n';
  z= 4 + subtraction (x,y);
  cout << "The fourth result is " << z << '\n';
  return 0;
}
```

The first result is 5

The second result is 5

The third result is 2

The fourth result is 6

In this case we have created a function called subtraction. The only thing that this function does is to subtract both passed parameters and to return the result. Nevertheless, if we examine function main we will see that we have made several calls to function subtraction. We have used some different calling methods so that you see other ways or moments when a function can be called.

In order to fully understand these examples you must consider once again that a call to a function could be replaced by the value that the function call itself is going to return. For example, the first case (that you should already know because it is the same pattern that we have used in previous examples):

```
z = subtraction (7,2);

cout << "The first result is " << z;
```

If we replace the function call by the value it returns (i.e., 5), we would have:

```
z = 5;

cout << "The first result is " << z;
```

As well as

```
cout << "The second result is " << subtraction (7,2);
```

has the same result as the previous call, but in this case we made the call to subtraction directly as an insertion parameter for cout. Simply consider that the result is the same as if we had written:

```
cout << "The second result is " << 5;
```

since 5 is the value returned by subtraction (7,2).

In the case of:

```
cout << "The third result is " << subtraction (x,y);
```

The only new thing that we introduced is that the parameters of subtraction are variables instead of constants. That is perfectly valid. In this case the values passed to function subtraction are the values of x and y, that are 5 and 3 respectively, giving 2 as result. The fourth case is more of the same. Simply note that instead of:

```
z = 4 + subtraction (x,y);
```

we could have written:

```
z = subtraction (x,y) + 4;
```

with exactly the same result. I have switched places so you can see that the semicolon sign (;) goes at the end of the whole statement. It does not necessarily have to go right after the function call. The explanation might be once again that you imagine that a function can be replaced by its returned value:

```
z = 4 + 2;

z = 2 + 4;
```

Types of Functions-

=>Functions with no return type and no argument. The use of void-

If you remember the syntax of a function declaration:

type name (argument1, argument2 ...) statement

you will see that the declaration begins with a type, that is the type of the function itself (i.e., the type of the datum that will be returned by the function with the return statement). But what if we want to return no value?

Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value. In this case we should use the void type specifier for the function. This is a special specifier that indicates absence of type.

```
// void function example

#include <iostream.h>

void printmessage ()

{

  cout << "I'm a function!";

}

int main ()

{

  printmessage ();

  return 0;

}
```

I'm a function!

void can also be used in the function's parameter list to explicitly specify that we want the function to take no actual parameters when it is called. For example, function printmessage could have been declared as:

```
void printmessage (void)

{

  cout << "I'm a function!";

}
```

Although it is optional to specify void in the parameter list. In C++, a parameter list can simply be left blank if we want a function with no parameters.

What you must always remember is that the format for calling a function includes specifying its name and enclosing its parameters between parentheses. The non-existence of parameters does not exempt us from the obligation to write the parentheses. For that reason the call to printmessage is:

printmessage ();

The parentheses clearly indicate that this is a call to a function and not the name of a variable or some other C++ statement. The following call would have been incorrect:

printmessage;

=>Function receiving argument and returning value

```
#include<iostream.h>

int  add(int , int);

void main()

{

int a=10,b=20,c;

c=add(a,b);

cout<<"Addition ="<<c;

}

int add(int  i, int j)

{

  return i+j;

}
```

it will return the output: Addition 30 .

=> A function receiving argument and no returning value

```
#include<iostream.h>

void sqrt(int);

void main()

{

int max;

cout<<"Enter value";

cin>>max;

for(int i=1; i<=max;i++)
```

```
  {
    sqrt(i);
  }
}
 void sqrt(int n)
{
   float value;
  value=n*n;
cout<<"The square of "<<n<< "is" <<value<<endl;
}
```

output is:

Enter value: 3

The square of 1 is 1

The square of 2 is 4

The square of 3 is 9

Types of Calling Function

1. Call by value
2. Call by Reference

Call by Value

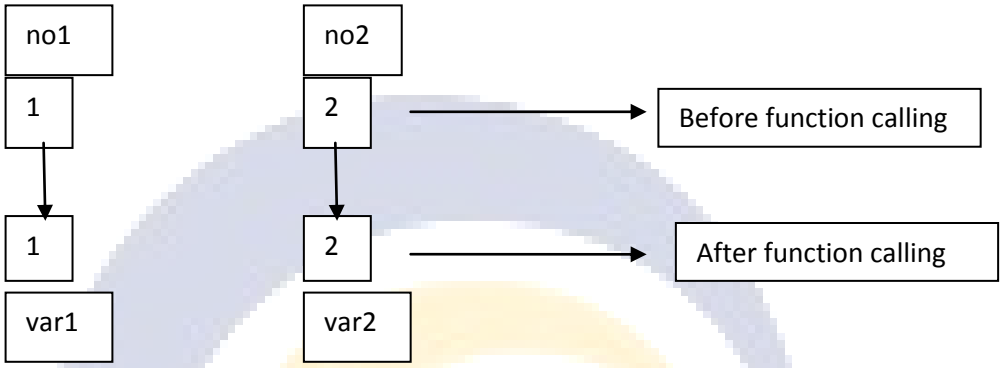
It makes the function more self-contained and protected from accidental alteration. This way is useful when you want that the calling function should not be in a position to change the value of original variables.

Eg.

```
#include<iostream.h>
int sum(int ,int);
void main()
{
int no1,no2,s;
cout<<"Enter two numbers";
cin>>no1>>no2;
s=sum(no1,no2);
cout<<no1;
cout<<no2;
cout<<s;
}
int sum(int var1, int var2)
{
```

```
cout<<var1<<endl;
cout<<var2<<endl;
return var1+var2;
}
```

The function does not access the original variable in the calling program,only the copy is created. This can be depicted using following figure:



Call By Reference

A reference provides an alias(a different name) for the same variable. Passing argument by reference is useful when the calling function need to modify the original variable,In this type, a reference of the original variable is passed to the function.

```
#include<iostream.h>
int sum(int & ,int &);
void main()
{
int no1,no2,s;
cout<<"Enter two numbers";
cin>>no1>>no2;
s=sum(no1,no2);
cout<<no1;
cout<<no2;
cout<<s;
}
int sum(int &var1, int &var2)
{
cout<<var1<<endl;
cout<<var2<<endl;
return var1+var2;
}
```

No1 no2

15 20

Var1 var2

Using this method alias names are created for original variables. If the value of var1 and var2 will be modified the original will also change.

Overloaded functions.

In C++ two different functions can have the same name if their parameter types or number are different. That

means that you can give the same name to more than one function if they have either a different number of

parameters or different types in their parameters. For example:

```
// overloaded function
#include <iostream>
using namespace std;
int operate (int a, int b)
{
  return (a*b);
}
float operate (float a, float b)
{
  return (a/b);
}
int main ()
{
  int x=5,y=2;
float n=5.0,m=2.0;
  cout << operate (x,y);
  cout << "\n";
  cout << operate (n,m);
  cout << "\n";
  return 0;
}
```

10

2.5

In this case we have defined two functions with the same name, operate, but one of them accepts two parameters of type int and the other one accepts them of type float. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two ints as its arguments it calls to the function that has two int parameters in its prototype and if it is called with two floats it will call to the one which has two float parameters in its prototype.

In the first call to operate the two arguments passed are of type int, therefore, the function with the first prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type float, so the function with the second prototype is called. This one has a different behavior: it divides one parameter by the other. So the behavior of a call to operate depends on the type of the arguments passed because the function has been overloaded.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

Inline functions.

The inline specifier indicates the compiler that inline substitution is preferred to the usual function call mechanism for a specific function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time.

The format for its declaration is:

inline type name (arguments ...) { instructions ... }

and the call is just like the call to any other function. You do not have to include the inline keyword when calling the function, only in its declaration.

Most compilers already optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.

Lesson 7

Class and Object

In object-oriented programming languages like C++, the data and functions (procedures to manipulate the data) are bundled together as a self-contained unit called an object. A class is an extended concept similar to that of structure in C programming language; this class describes the data properties alone. In C++ programming language, class describes both the properties (data) and behaviors (functions) of objects. Classes are not objects, but they are used to instantiate objects.

Features of Class:

Classes contain data known as members and member functions. As a unit, the collection of members and member functions is an object. Therefore, this unit of objects makes up a class.

How to write a Class:

In Structure in C programming language, a structure is specified with a name. The C++ programming language extends this concept. A class is specified with a name after the keyword class.

The starting flower brace symbol '{'is placed at the beginning of the code. Following the flower brace symbol, the body of the class is defined with the member functions data. Then the class is closed with a flower brace symbol '}' and concluded with a colon ';'.

```
class   one

{

  data;

  member_functions;

}
```


Creation of Objects:

Once the class is created, one or more objects can be created from the class as objects are instance of the class.

Just as we declare a variable of data type int as:

int x;

Objects are also declared as:

Class_name followed by object_name;

Example:

One o1;

This declares e1 to be an object of class one.

For example a complete class and object declaration is given below:

```
class one
{
  private:
  int x,y;
  public:
  void sum()
  {
    ....
    ....
  }
};

void main()
{
  one o1;
  ....
  ....
}
```

Encapsulation

Encapsulation is also called protection or information hiding. In fact, encapsulation, protection and information hiding are three overlapping concepts.

Introduction

Encapsulation is the process of combining data and functions into a single unit called class. Using the method of encapsulation, the programmer cannot directly access the data. Data is only accessible through the functions existing inside the class. Data encapsulation led to the important concept of data hiding. Data hiding is the implementation details of a class that are hidden from the user. The concept of restricted access led programmers to write specialized functions or methods for performing the operations on hidden members of the class. Attention must be paid to ensure that the class is designed properly.

Protection and information hiding are techniques used to accomplish encapsulation of an object. Protection is when you limit the use of class data or methods. Information hiding is when you remove data, methods or code from a class's public interface in order to refine the scope of an object.

```
class A

{

public:

integer d;

};
```

If you moved that data member from the public scope of the private scope, then you would be hiding the member. Better said, you are hiding the member from the public

interface.


```
class A

{

private:

integer d;

};
```

Data Abstraction

Data abstraction refers to, providing only essential information to the outside word and hiding their background details ie. to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players BUT you do not know it's internal detail that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen. Thus we can say, a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.

C++ classes provides great level of data abstraction. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data ie. state without actually knowing how class has been implemented internally.

For example, your program can make a call to the sort() function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

In C++ we use classes to define our own abstract data types (ADT). You can use the cout object of class ostream to stream data to standard output like this:

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

int main( )
{
   cout << "Hello C++" <<endl;
   return 0;
}
```


Access Labels Enforce Abstraction:

In C++ we use access labels to define the abstract interface to the class. A class may contain zero or more access labels:

- Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.

- Members defined with a private label are not accessible to code that uses the class. The private sections hides the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction:

Data abstraction provide two important advantages:

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data are public, then any function that directly accesses the data members of the old representation might be broken.

Lesson 8

constructors and destructors

Because classes have complicated internal structures, including data and functions, object initialization and cleanup for classes is much more complicated than it is for simple data structures. Constructors and destructors are special member functions of classes that are used to construct and destroy class objects. Construction may involve memory allocation and initialization for objects. Destruction may involve cleanup and deallocation of memory for objects.

Like other member functions, constructors and destructors are declared within a class declaration. They can be defined inline or external to the class declaration. Constructors can have default arguments. Unlike other member functions, constructors can have member initialization lists. The following restrictions apply to constructors and destructors:

- Constructors and destructors do not have return types nor can they return values.
- References and pointers cannot be used on constructors and destructors because their addresses cannot be taken.
- Constructors cannot be declared with the keyword virtual.
- Constructors and destructors cannot be declared static, const, or volatile.
- Unions cannot contain class objects that have constructors or destructors.

Constructors and destructors obey the same access rules as member functions. For example, if you declare a constructor with protected access, only derived classes and friends can use it to create class objects.The compiler automatically calls constructors when defining class objects and calls destructors when class objects go out of scope. A constructor does not allocate memory for the class object its this pointer refers to, but may allocate storage for more objects than its class object refers to. If memory allocation is required for objects, constructors can explicitly call the new operator. During cleanup, a destructor may release objects allocated by the corresponding constructor. To release objects, use the delete operator.

Derived classes do not inherit or overload constructors or destructors from their base classes, but they do call the constructor and destructor of base classes. Destructors can be declared with the keyword virtual. Constructors are also called when local or temporary class objects are created, and destructors are called when local or temporary objects go out of scope.

Types of constructor

1. **Default Constructor**

These types of constructors have no parameter. It is called automatically by the system when an object is created.

```
#include<iostream.h>

#include<conio.h>

class counter

        {

                int count;

        public:

                counter()

                {

                count=0;

                }

                void increment()

                {

                count++;

                }

                int getcount()

                {

                return count;

                }

        };

void main()

{

        counter c1,c2;
```

```
                cout<<"\n c1="<<c1.getcount();
                cout<<"\n c2="<<c2.getcount();
                c1.increment();
                c2.increment();
    cout<<"\n c1="<<c1.getcount();
                cout<<"\n c2="<<c2.getcount();
                getch();
}
```

Output:

C1=0

C2=0

C1=1

C2=1

2. Parameterize or overloaded constructor

As like function overloading we can overload constructor also. This type of constructor has parameters. When the object of class is created with parameters , constructor with parameter is invoked automatically otherwise default constructor is invoked.

```
#include<iostream.h>
#include<conio.h>
class student
{
        int roll;
        float marks;
        public:
```

```
                student()
                {       roll=101;
                        marks=50;       }
                student(int r, float m)
                {
                roll=r;
                marks=m;
                }
                void print()
                {
                        cout<<roll<<" "<<marks;
                }
};
void main()
{
  student stud1, stud2(105,99);
  stud1.print();
  cout<<endl;
  stud2.print();
  getch();}
```

Output:

101 50

105 99

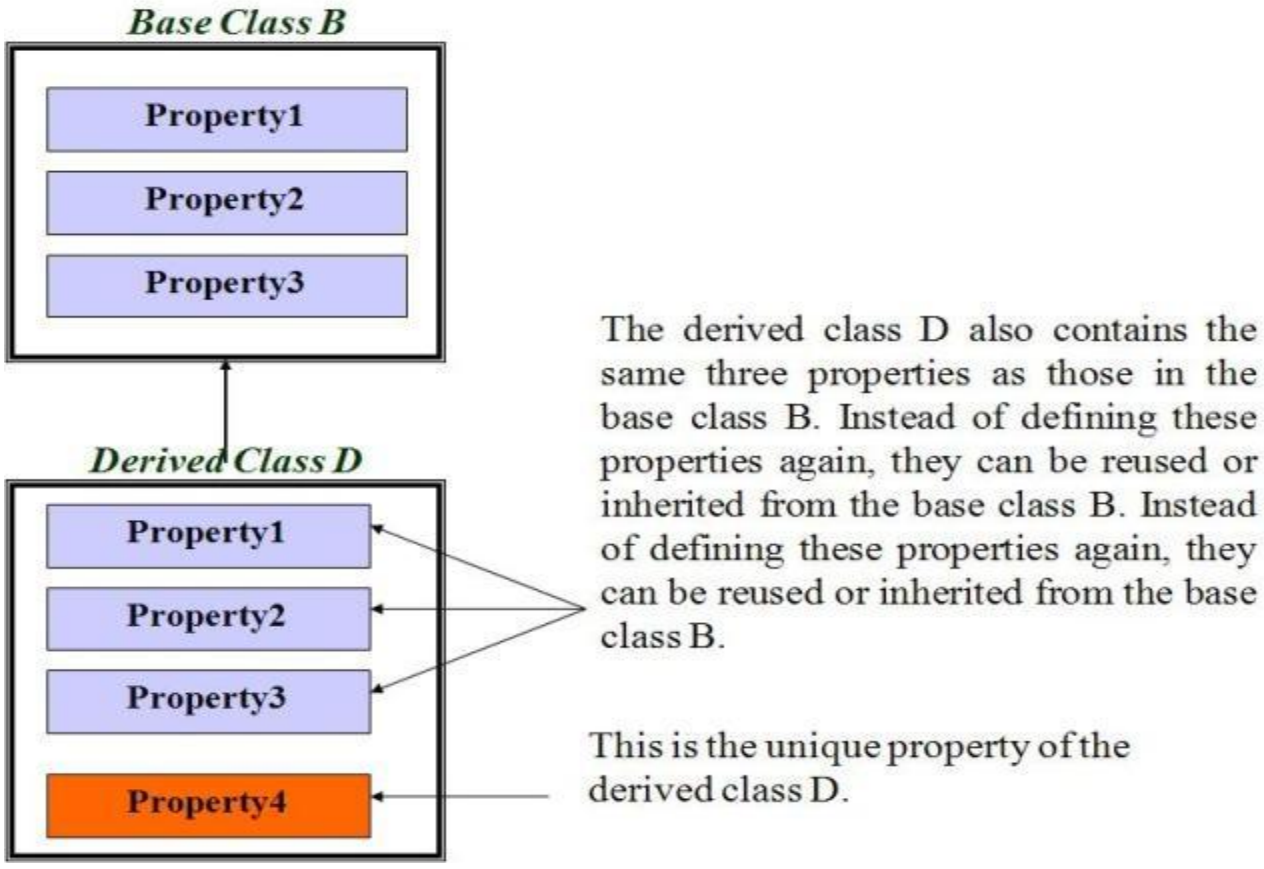
LESSON-9

INHERITANCE

Inheritance is the process of creating new classes from the existing class or classes.

Using inheritance, one can create general class that defines traits common to a set of related items. This class can then be inherited (reused) by the other classes by using the properties of the existing ones with the addition of its own unique properties.

The old class is referred to as the base class and the new classes, which are inherited from the base class, are called derived classes.



In above figure, the class B contains the three member i.e. Property1, Property2 and Property3.

The second class named as D contains four members i.e. Property1, Property2, Property3 and Property4.

Out of these, three members (Property1, Property2, Property3) of the class D are same as that of class B, while the one member i.e. Property4 is unique.

In the class D, instead of defining all the three members again, one can reuse them from the class B. This feature is known as inheritance which saves time, space, money and increases the reliability and efficiency.

Forms of Inheritance

Single Inheritance If a class is derived from a single base class, it is called as single inheritance.

Multiple Inheritance If a class is derived from more than one base class, it is known as multiple inheritance

Multilevel Inheritance The classes can also be derived from the classes that are already derived. This type of inheritance is called multilevel inheritance.

Hierarchical Inheritance If a number of classes are derived from a single base class, it is called as hierarchical inheritance.

A derived class can be defined as follows:

```
class derived_class_name : access_specifier base_class_name
{
data members of the derived class ;
member functions of the derived class ;
}
```

The colon (:), indicates that the class derived_class_name is derived from the class base_class_name.
The access_specifier may be public, private or protected (will be discussed further). If no access_specifier is specified, it is private by default.
The access_specifier indicates whether the members of the base class are privately derived or publicly derived.

Public inheritance

When a derived class publicly inherits the base class, all the public members of the base class also become public to the derived class and the objects of the derived class can access the public members of the base class. The following program will illustrate the use of the single inheritance.

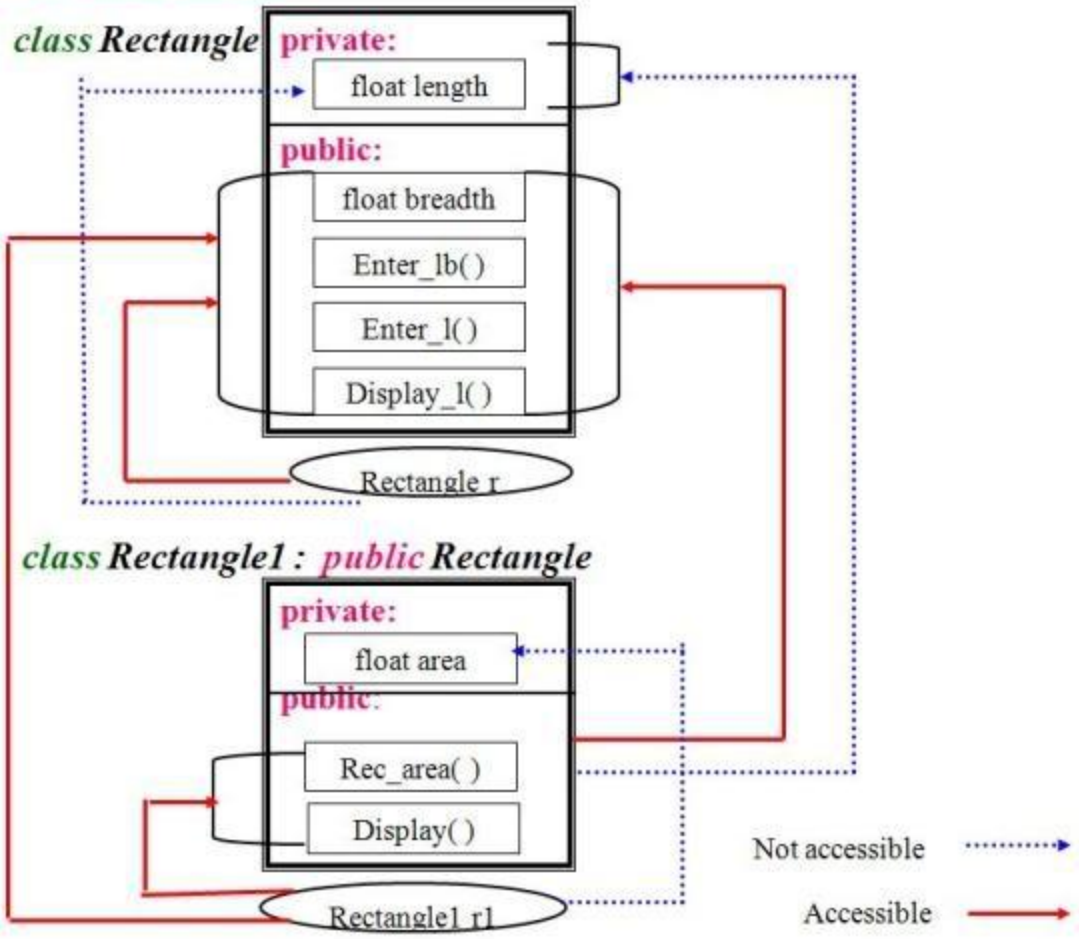
This program has a base class B, from which class D is inherited publicly.

Example:

```
#include
class Rectangle
{
private:
float length ; // This can't be inherited
public:
float breadth ; // The data and member functions are inheritable
void Enter_lb(void)
{
cout << "\n Enter the length of the rectangle : ";
cin >> length ;
cout << "\n Enter the breadth of the rectangle : ";
cin >> breadth ;
}
float Enter_l(void)
{ return length ; }
}; // End of the class definition

class Rectangle1 : public Rectangle
{
private:
float area ;
public:
void Rec_area(void)
{ area = Enter_l( ) * breadth ; }
// area = length * breadth ; can't be used here

void Display(void)
{
cout << "\n Length = " << Enter_l( ) ; // Object of the derived class can't
// inherit the private member of the base class. Thus the member
// function is used here to get the value of data member 'length'.
cout << "\n Breadth = " << breadth ;
cout << "\n Area = " << area ;
}
}; // End of the derived class definition D
void main(void)
{
Rectangle1 r1 ;
r1.Enter_lb( );
r1.Rec_area( );
r1.Display( );
}
```

Figure- Public Inheritance

Fig: public inheritance



Private inheritance

When a derived class privately inherits a base class, all the public members of the base class become private for the derived class.

In this case, the public members of the base class can only be accessed by the member functions of the derived class.

The objects of the derived class cannot access the public members of the base class. Note that whether the derived class is inherited publicly or privately from the base class, the private members of the base class cannot be inherited.

Example:

```
#include
#include
```

```
class Rectangle
{
int length, breadth;
public:
void enter()
{
cout << "\n Enter length: "; cin >> length;
cout << "\n Enter breadth: "; cin >> breadth;
}
int getLength()
{
return length;
}
int getBreadth()
{
return breadth;
}
void display()
{
cout << "\n Length= " << length;
cout << "\n Breadth= " << breadth;
}
};

class RecArea : private Rectangle
{

public:
void area_rec()
{
enter();
cout << "\n Area = " << (getLength() * getBreadth());
}
};
void main()
{
clrscr();
RecArea r ;
r.area_rec();
getch();
}
```

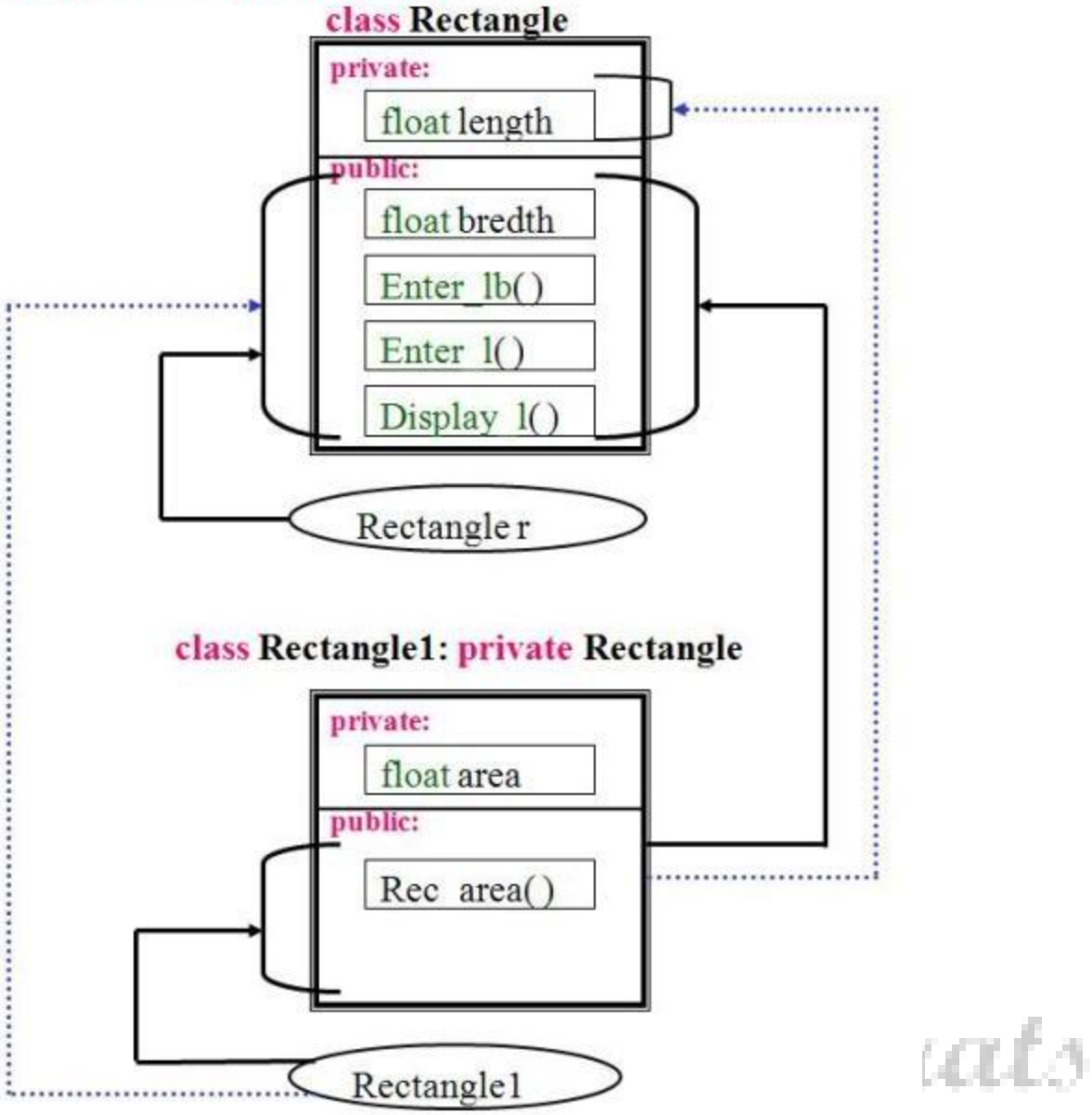
Example:2

```
#include
class Rectangle // Base class
{
private:
```

```
float length ; // This data member can't be inherited
public:
float breadth ; // This data member is inheritable
void Enter_lb(void)
{
cout << "\n Enter the length of the rectangle: "; cin >> length ;
cout << "\n Enter the breadth of the rectangle: "; cin >> bredth ;
}
float Enter_l(void)
{
return length ;
}
void Display_l(void)
{
cout << "\n Length = " << length ;
}
}; // End of the base class Rectangle.
// Defining the derived class Rectangle1. This class has been derived from the
// base class i.e. Rectangle, privately.
class Rectangle1 : private Rectangle // All the public members of the base class
{ // Rectangle become private for the derived class Rectangle1.
private:
float area ;
public:
void Rec_area(void)
{
Enter_lb( );
area = Enter_l( ) * bredth ; // length can't be used directly
}
void Display(void)
{
Display_l( ); // Displays the value of length.
cout << "\n Bredth = " << bredth ;
cout << "\n Area = " << area << endl ;
}
};
void main(void)
{
Rectangle1 r1 ;
r1.Rec_area( ) ;
// r.Enter_lb( ); will not work as it has become private for the derived class.
r1.Display( ) ;
// r.Display_l( ) will not work as it also has become private for the derived class.
}
```

Fig: private inheritance

Fig: private inheritance



The protected access specifier

The third access specifier provided by C++ is protected.

The members declared as protected can be accessed by the member functions within their own class and any other class immediately derived from it.

These members cannot be accessed by the functions outside these two classes.

Therefore, the objects of the derived class cannot access protected members of the

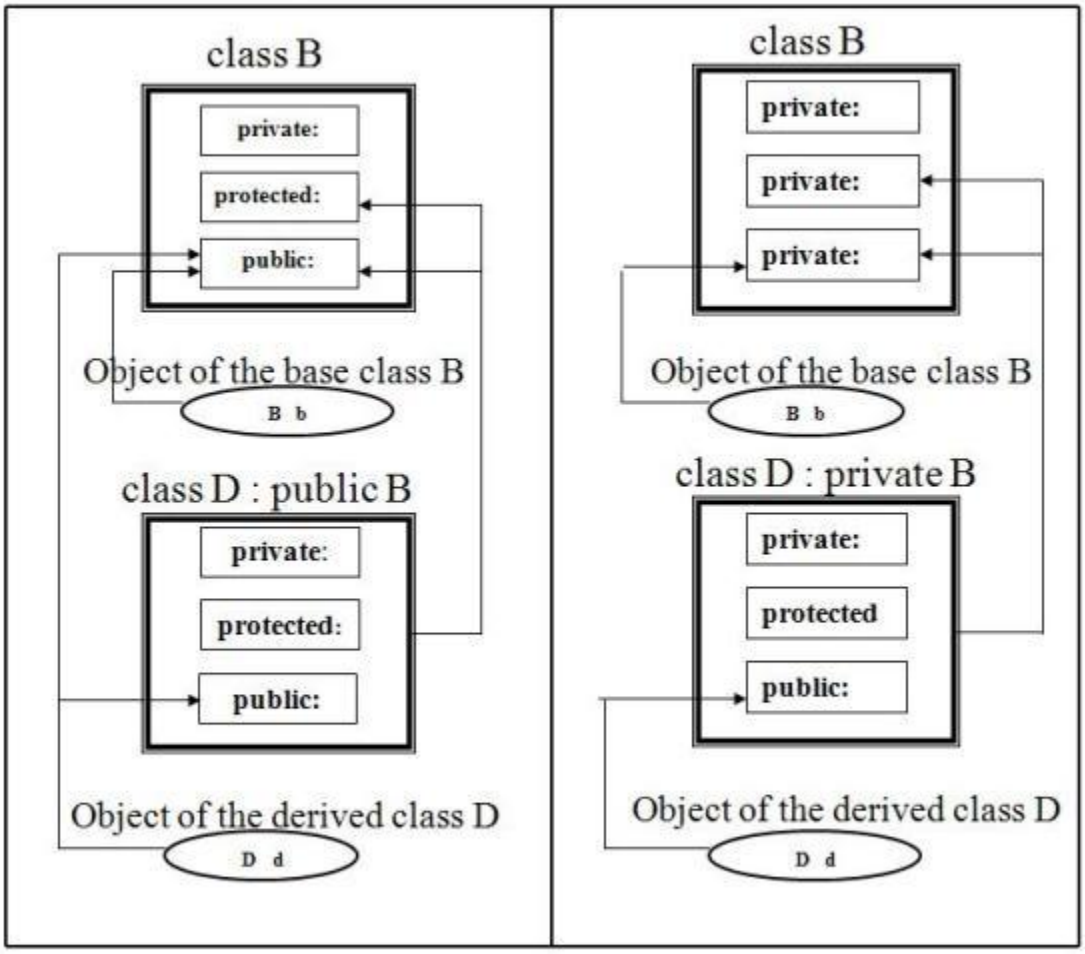
base class.

When the protected members (data, functions) are inherited in public mode, they become protected in the derived class. Thus, they can be accessed by the member functions of the derived class.

On other hand, if the protected members are inherited in the private mode, the members also become private in the derived class.

They can also be accessed by the member functions of the derived class, but cannot be inherited further.

Fig: Access specifier with inheritance



Overriding the member functions:

The member functions can also be used in a derived class, with the same name as those in the base class.

One might want to do this so that calls in the program work the same way for objects of both base and derived classes.

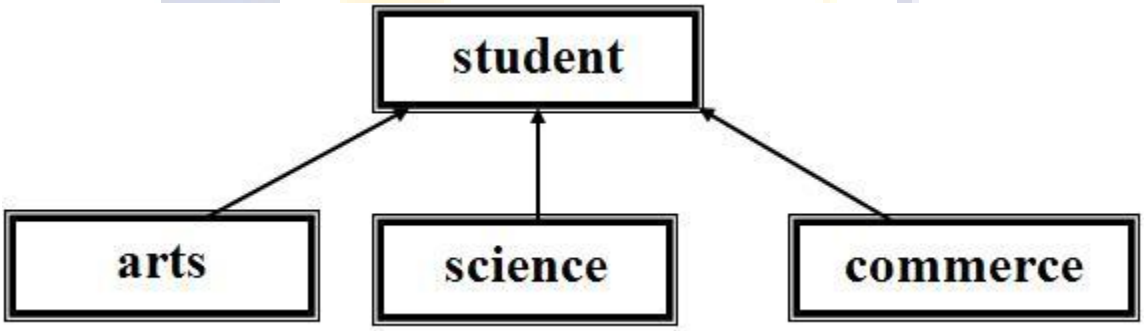
The following program will illustrate this concept.

```
#include
const int len = 20 ;
class Employee
{
private:
char F_name[len];
int I_number ;
int age ;
float salary ;
public:
void Enter_data(void)
{
cout << "\n Enter the first name = " ; cin >> F_name ;
cout << "\n Enter the identity number = " ; cin >> I_number ;
cout << "\n Enter the age = " ; cin >> age ;
cout << "\n Enter the salary = " ; cin >> salary ;
}
void Display_data(void)
{
cout << "\n Name = " << F_name ;
cout << "\n Identity Number = " << I_number ;
cout << "\n Age = " << age ;
cout << "\n Salary = " << salary ;
}
}; // End of the base class
class Engineer : public Employee
{
private:
char design[len] ; // S_Engineer, J_Engineer, Ex_Engineer etc

public:
void Enter_data( )
{
Employee :: Enter_data( ) ; // Overriding of the member function
cout << "\n Enter the designation of the Engineer: " ; cin >> design ;
}
```

```
void Display_data(void)
{
cout << "\n *******Displaying the particulars of the Engineer**** \n" ;
Employee :: Display_data( ) ; // Overriding of the member function
cout << "\n Designition = " << design ;
}
}; // End of the derived class

void main(void)
{
Engineer er ;
er.Enter_data( ) ;
er.Display_data( );
}
```

Hierarchical Inheritance:

When two or more classes are derived from a single base class, then Inheritance is called the hierarchical inheritance. The representation of the hierarchical inheritance is shown in the following Fig.



In the above Fig., student is a base class, from which the three classes viz. arts, science and commerce have been derived. Now, let us write a program that illustrates the hierarchical inheritance, based on the above design.

Example

```
include
const int len = 20 ;
class student // Base class
{
private: char F_name[len] , L_name[len] ;
int age, int roll_no ;
public:
void Enter_data(void)
```

```
{
cout << "\n\t Enter the first name: " ; cin >> F_name ;
cout << "\t Enter the second name: "; cin >> L_name ;
cout << "\t Enter the age: " ; cin >> age ;
cout << "\t Enter the roll_no: " ; cin >> roll_no ;
}
void Display_data(void)
{
cout << "\n\t First Name = " << F_name ;
cout << "\n\t Last Name = " << L_name ;
cout << "\n\t Age = " << age ;
cout << "\n\t Roll Number = " << roll_no ;
}
};
class arts : public student
{
private:
char asub1[len] ;
char asub2[len] ;
char asub3[len] ;
public:
void Enter_data(void)
{
student :: Enter_data( );
cout << "\t Enter the subject1 of the arts student: "; cin >> asub1 ;
cout << "\t Enter the subject2 of the arts student: "; cin >> asub2 ;
cout << "\t Enter the subject3 of the arts student: "; cin >> asub3 ;
}
void Display_data(void)
{
student :: Display_data( );
cout << "\n\t Subject1 of the arts student = " << asub1 ;
cout << "\n\t Subject2 of the arts student = " << asub2 ;
cout << "\n\t Subject3 of the arts student = " << asub3 ;
}
};
class commerce : public student
{
private: char csub1[len], csub2[len], csub3[len] ;
public:
void Enter_data(void)
{
student :: Enter_data( );
cout << "\t Enter the subject1 of the commerce student: ";
cin >> csub1;
cout << "\t Enter the subject2 of the commerce student: ";
cin >> csub2 ;
```

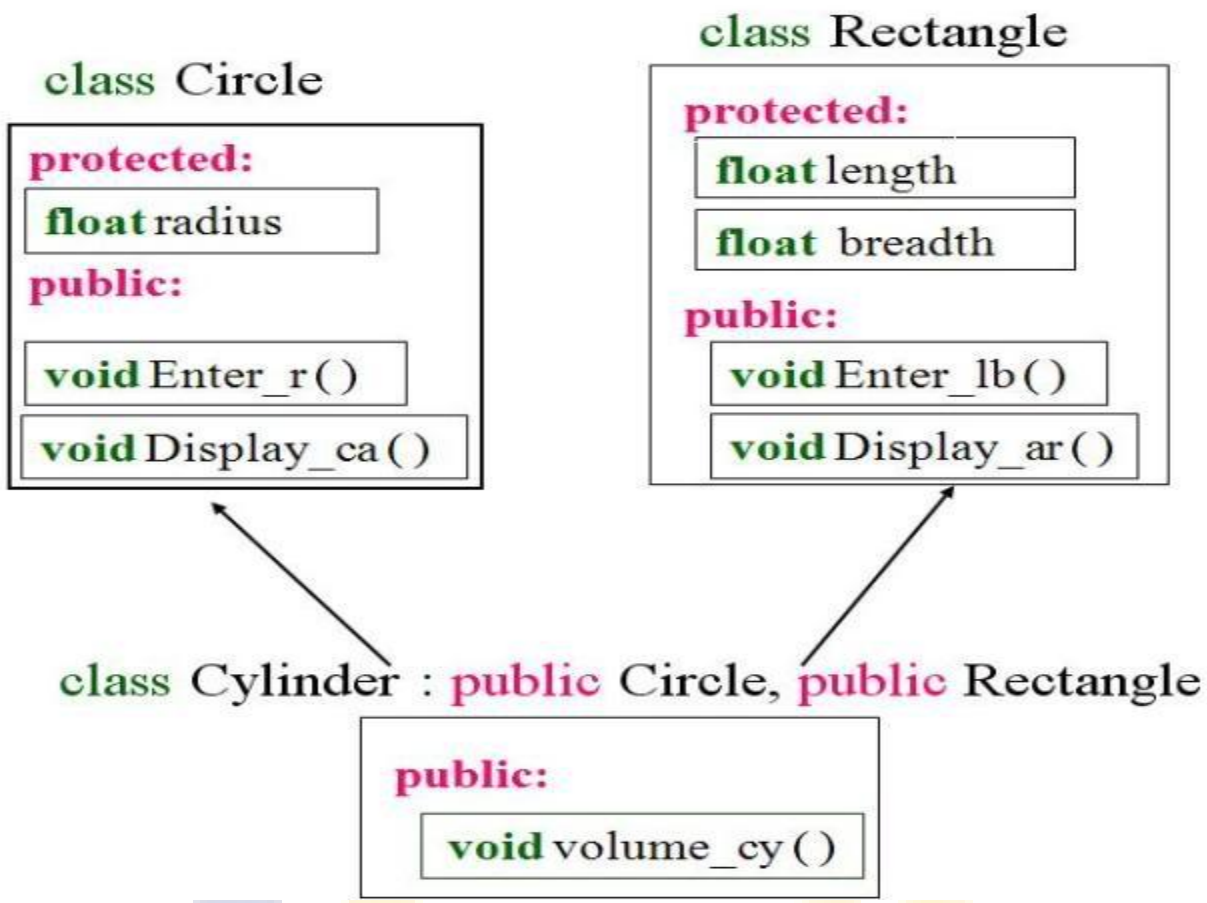
```
cout << "\t Enter the subject3 of the commerce student: ";
cin >> csub3 ;
}
void Display_data(void)
{
student :: Display_data( );
cout << "\n\t Subject1 of the commerce student = " << csub1 ;
cout << "\n\t Subject2 of the commerce student = " << csub2 ;
cout << "\n\t Subject3 of the commerce student = " << csub3 ;
}
};

void main(void)
{
arts a ;
cout << "\n Entering details of the arts student\n" ;
a.Enter_data( );
cout << "\n Displaying the details of the arts student\n" ;
a.Display_data( );
science s ;
cout << "\n\n Entering details of the science student\n" ;
s.Enter_data( );
cout << "\n Displaying the details of the science student\n" ;
s.Display_data( );
commerce c ;
cout << "\n\n Entering details of the commerce student\n" ;
c.Enter_data( );
cout << "\n Displaying the details of the commerce student\n";
c.Display_data( );
}
```

Multiple Inheritance

When a class is inherited from more than one base class, it is known as multiple inheritance.
The syntax for defining a subclass, which is inheriting more than one classes is:

```
class Subclass : access_specifier Baseclass1,
access_specifier Baseclass2, ……
…….. access_specifier Baseclass_n
{
members of the derived class ;
};
```

The following figure illustrates the use of multiple inheritance.



In the above figure, Circle and Rectangle are two base classes from which the class Cylinder is being inherited.

The data members of both the base classes are declared in protected mode. Thus, the class Cylinder can access the data member radius of class Circle and data member length, breadth of the class Rectangle, but the objects of the class Cylinder cannot access these protected data members.

The volume of the cylinder is equal to 22/7*(radius*radius*length). Thus, instead of defining these data again, they can be inherited from the base classes Circle and Rectangle (radius from class Circle and length from class Rectangle).

Example:

```
#include
class Circle // First base class
{
protected:
float radius ;
public:
```

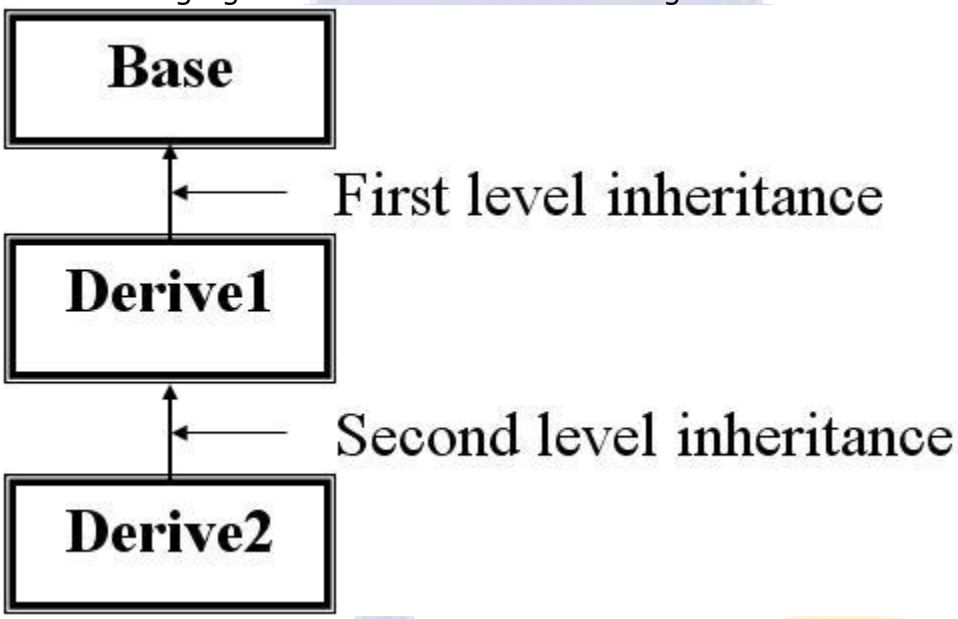
```
void Enter_r(void)
{
cout << "\n\t Enter the radius: "; cin >> radius ;
}
void Display_ca(void)
{
cout << "\t The area = " << (22/7 * radius*radius) ;
}
};
class Rectangle // Second base class
{
protected:
float length, breadth ;
public:
void Enter_lb(void)
{
cout << "\t Enter the length : "; cin >> length ;
cout << "\t Enter the breadth : " ; cin >> breadth ;
}
void Display_ar(void)
{
cout << "\t The area = " << (length * breadth);
}
};
class Cylinder : public Circle, public Rectangle // Derived class, inherited
{ // from classes Circle & Rectangle
public:
void volume_cy(void)
{
cout << "\t The volume of the cylinder is: "
<< (22/7* radius*radius*length) ;
}
};
void main(void)
{
Circle c ;
cout << "\n Getting the radius of the circle\n" ;
c.Enter_r( );
c.Display_ca( );
Rectangle r ;
cout << "\n\n Getting the length and breadth of the rectangle\n\n";
r.Enter_l( );
r.Enter_b( );
r.Display_ar( );
Cylinder cy ; // Object cy of the class cylinder which can access all the
// public members of the class circle as well as of the class rectangle
cout << "\n\n Getting the height and radius of the cylinder\n";
```

```
cy.Enter_r( );
cy.Enter_lb( );
cy.volume_cy( );
}
```

Multilevel Inheritance:

It has been discussed so far that a class can be derived from a class.
C++ also provides the facility of multilevel inheritance, according to which the derived class can also be derived by an another class, which in turn can further be inherited by another and so on.

The following figure will illustrate the meaning of the multilevel inheritance.



In the above figure, class B represents the base class. The class D1 that is called first level of inheritance, inherits the class B. The derived class D1 is further inherited by the class D2, which is called second level of inheritance.

Example: Multilevel Inheritance

```
#include
class Base
{
protected:
int b;
public:
void EnterData( )
{
cout << "\n Enter the value of b: ";
cin >> b;
}
```

```
void DisplayData( )
{
cout << "\n b = " << b;
}
};
class Derive1 : public Base
{
protected:
int d1;
public:
void EnterData( )
{
Base:: EnterData( );
cout << "\n Enter the value of d1: ";
cin >> d1;
}
void DisplayData( )
{
Base::DisplayData();
cout << "\n d1 = " << d1;
}
};
class Derive2 : public Derive1
{
private:
int d2;
public:
void EnterData( )
{
Derive1::EnterData( );
cout << "\n Enter the value of d2: ";
cin >> d2;
}
void DisplayData( )
{
Derive1::DisplayData( );
cout << "\n d2 = " << d2;
}
};
int main( )
{
Derive2 objd2;
objd2.EnterData( );
objd2.DisplayData( );
return 0;
}
```

Lesson 10

Polymorphism

Before getting into this section, it is recommended that you have a proper understanding of pointers and class inheritance. If any of the following statements seem strange to you, you should review the indicated sections:

Statement:	Explained in:
int a::b(int c) { }	Classes
a->b	Data Structures
class a: public b { };	Friendship and inheritance

Pointers to base class

One of the key features of derived classes is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature, that brings Object Oriented Methodologies to its full potential.
We are going to start by rewriting our program about the rectangle and the triangle of the previous section taking into consideration this pointer compatibility property:

```
// pointers to base class
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
  };

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

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2);
}
  };

int main () {
```

```
20
10
```

```
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

In function main, we create two pointers that point to objects of class CPolygon (ppoly1 and ppoly2). Then we assign references to rect and trgl to these pointers, and because both are objects of classes derived from CPolygon, both are valid assignment operations.

The only limitation in using *ppoly1 and *ppoly2 instead of rect and trgl is that both *ppoly1 and *ppoly2 are of type CPolygon* and therefore we can only use these pointers to refer to the members that CRectangle and CTriangle inherit from CPolygon. For that reason when we call the area() members at the end of the program we have had to use directly the objects rect and trgl instead of the pointers *ppoly1 and *ppoly2.

In order to use area() with the pointers to class CPolygon, this member should also have been declared in the class CPolygon, and not only in its derived classes, but the problem is that CRectangle and CTriangle implement different versions of area, therefore we cannot implement it in the base class. This is when virtual members become handy:

Virtual members

A member of a class that can be redefined in its derived classes is known as a virtual member. In order to declare a member of a class as virtual, we must precede its declaration with the keyword virtual:

```
// virtual members
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return (0); }
  };

```

```
20
10
0
```

```
class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2);
}
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon poly;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  CPolygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  cout << ppoly3->area() << endl;
  return 0;
}
```

Now the three classes (CPolygon, CRectangle and CTriangle) have all the same members: width, height, set_values() and area().

The member function area() has been declared as virtual in the base class because it is later redefined in each derived class. You can verify if you want that if you remove this virtual keyword from the declaration of area() within CPolygon, and then you run the program the result will be 0 for the three polygons instead of 20, 10 and 0. That is because instead of calling the corresponding area() function for each object (CRectangle::area(), CTriangle::area() and CPolygon::area(), respectively), CPolygon::area() will be called in all cases since the calls are via a pointer whose type is CPolygon*.

Therefore, what the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class but is pointing to an object of the derived class, as in the above example.

A class that declares or inherits a virtual function is called a polymorphic class.

Note that despite of its virtuality, we have also been able to declare an object of type CPolygon and to call its own area() function, which always returns 0.

Abstract base classes

Abstract base classes are something very similar to our CPolygon class of our previous example. The only difference is that in our previous example we have defined a valid area() function with a minimal functionality for objects that were of class CPolygon (like the object poly), whereas in an abstract base classes we could leave that area() member function without implementation at all. This is done by appending =0 (equal to zero) to the function declaration.

An abstract base CPolygon class could look like this:

```
// abstract class CPolygon
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area () =0;
};
```

Notice how we appended =0 to virtual int area () instead of specifying an implementation for the function. This type of function is called a pure virtual function, and all classes that contain at least one pure virtual function are abstract base classes.

The main difference between an abstract base class and a regular polymorphic class is that because in abstract base classes at least one of its members lacks implementation we cannot create instances (objects) of it.

But a class that cannot instantiate objects is not totally useless. We can create pointers to it and take advantage of all its polymorphic abilities. Therefore a declaration like:

```
CPolygon poly;
```

would not be valid for the abstract base class we have just declared, because tries to instantiate an object. Nevertheless, the following pointers:

```
1 CPolygon * ppoly1;
2 CPolygon * ppoly2;
```

would be perfectly valid.

This is so for as long as CPolygon includes a pure virtual function and therefore it's an abstract base class. However, pointers to this abstract base class can be used to point to objects of derived classes.

Here you have the complete example:

```
// abstract base class
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
  };

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

class CTriangle: public CPolygon {
  public:
    int area (void)
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  return 0;
}
```

```
20
10
```

If you review the program you will notice that we refer to objects of different but related classes using a unique type of pointer (CPolygon*). This can be tremendously useful. For example, now we can create a function member of the abstract base class CPolygon that is able to print on screen the result of the area() function even though CPolygon itself has no implementation for this function:

```
// pure virtual members can be called
// from the abstract base class
#include <iostream>
using namespace std;


class CPolygon {
 protected:
   int width, height;
 public:
   void set_values (int a, int b)
     { width=a; height=b; }
   virtual int area (void) =0;
   void printarea (void)
     { cout << this->area() << endl; }
 };


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


class CTriangle: public CPolygon {
 public:
   int area (void)
     { return (width * height / 2); }
 };
```

```
20
10
```

```
int main () {
 CRectangle rect;
 CTriangle trgl;
 CPolygon * ppoly1 = &rect;
 CPolygon * ppoly2 = &trgl;
 ppoly1->set_values (4,5);
 ppoly2->set_values (4,5);
 ppoly1->printarea();
 ppoly2->printarea();
 return 0;
}
```

Virtual members and abstract classes grant C++ the polymorphic characteristics that make object-oriented programming such a useful instrument in big projects. Of course, we have seen very simple uses of these features, but these features can be applied to arrays of objects or dynamically allocated objects.

Let's end with the same example again, but this time with objects that are dynamically allocated:

```
// dynamic allocation and polymorphism
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
    void printarea (void)
      { cout << this->area() << endl;
}
  };

class CRectangle: public CPolygon {
  public:
```

```
20
10
```

```
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
      { return (width * height / 2); }
  };

int main () {
  CPolygon * ppoly1 = new CRectangle;
  CPolygon * ppoly2 = new CTriangle;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```

Notice that the `ppoly` pointers:

```
1 CPolygon * ppoly1 = new CRectangle;
2 CPolygon * ppoly2 = new CTriangle;
```

are declared being of type pointer to `CPolygon` but the objects dynamically allocated have been declared having the derived class type directly.

Polymorphism

Polymorphism is the ability to use an operator or method in different ways. Polymorphism gives different meanings or functions to the operators or methods. Poly, referring to many, signifies the many uses of these operators and methods. A single method usage or an operator functioning in many ways can be called polymorphism. Polymorphism refers to codes, operations or objects that behave differently in different contexts.

The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance. C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes:

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

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape{
   public:
      Rectangle( int a=0, int b=0)
      {
        Shape(a, b);
      }
      int area ()
      {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
class Triangle: public Shape{
   public:
      Triangle( int a=0, int b=0)
      {
        Shape(a, b);
      }
      int area ()
      {
         cout << "Rectangle class area :" <<endl;
         return (width * height / 2);
      }
};
// Main function for the program
int main( )
{
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // store the address of Rectangle
   shape = &rec;
   // call rectangle area.
   shape->area();

   // store the address of Triangle
```

```
   shape = &tri;
   // call triangle area.
   shape->area();

   return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
Parent class area
Parent class area
```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called static resolution of the function call, or static linkage - the function call is fixed before the program is executed. This is also sometimes called early binding because the area() function is set during the compilation of the program. But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword virtual so that it looks like this:

```
class Shape
 {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      virtual int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

After this slight modification, when the previous example code is compiled and executed, it produces following result:

```
Rectangle class area
Triangle class area
```

This time the compiler looks at the contents of the pointer instead of it's type. Hence since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how polymorphism is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

Operator Overloading

If you specify more than one definition for a function name or an operator in the same scope, you have *overloaded* that function name or operator. Overloaded functions and operators are described in Overloading functions (C++ only) and Overloading operators (C++ only), respectively. Operator overloading is the ability to tell the compiler how to perform a certain operation when its corresponding operator is used on one or more variables. For example, the compiler acts differently with regards to the subtraction operator "-" depending on how the operator is being used. When it is placed on the left of a numeric value such as -48, the compiler considers the number a negative value. When used between two integral values, such as 80-712, the compiler applies the subtraction operation. When used between an integer and a double-precision number, such as 558-9.27, the compiler subtracts the left number from the right number; the operation produces a double-precision number. When the - symbol is doubled and placed on one side of a variable, such as --Variable or Variable--, the value of the variable needs to be decremented; in other words, the value 1 shall be subtracted from it. All of these operations work because the subtraction operator "-" has been reconfigured in various classes to act appropriately

An *overloaded declaration* is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different type.

```
#include<iostream.h>

class space

{

int x;

int y;

int 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<<"\n";
}
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;

}
```

Virtual Function:

A virtual function is a function in a base class that is declared using the keyword virtual. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as dynamic linkage, or late binding.

Pure Virtual Functions:

It's possible that you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      // pure virtual function
      virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called pure virtual function.

Friend Functions

A friend function is a fun ction that is not a member of a class but has access to the class's private and protected members. Friend functions are not considered class members; they are normal external functions that are given special access privileges. Friends are not in the class's scope, and they are not called using the member-selection operators (. and –>) unless they are members of another class. A friend function is declared by the class that is granting access. The friend declaration can be placed anywhere in the class declaration. It is not affected by the access control keywords.

The following example shows a Point class and a friend function, ChangePrivate. The friend function has access to the private data member of the Point object it receives as a parameter.

Example

Copy

```
// friend_functions.cpp

// compile with: /EHsc

#include <iostream>


using namespace std;

class Point

{

    friend void ChangePrivate( Point & );

public:

    Point( void ) : m_i(0) {}

    void PrintPrivate( void ){cout << m_i << endl; }


private:

    int m_i;

};


void ChangePrivate ( Point &i ) { i.m_i++; }
```

```
int main()
{
   Point sPoint;
   sPoint.PrintPrivate();
   ChangePrivate(sPoint);
   sPoint.PrintPrivate();
}
```

Output

```
0
1
```

A friend function of a class is defined outside that class's scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend as follows:

Lesson 11

Exception Handling

Exceptions

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in our program by transferring control to special functions called handlers.

To catch exceptions we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a try block. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

An exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block:

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

int main () {
  try
  {
    throw 20;
  }
  catch (int e)
  {
    cout << "An exception occurred.
Exception Nr. " << e << endl;
  }
  return 0;
}
```

```
An exception occurred. Exception Nr.
20
```

The code under exception handling is enclosed in a try block. In this example this code simply throws an exception:

```
throw 20;
```

A throw expression accepts one parameter (in this case the integer value 20), which is passed as an argument to the exception handler.

The exception handler is declared with the catch keyword. As you can see, it follows immediately the closing brace of the try block. The catch format is similar to a regular function that always has at least one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught.

We can chain multiple handlers (catch expressions), each one with a different parameter type. Only the handler that matches its type with the argument specified in the throw statement is executed.

If we use an ellipsis (...) as the parameter of catch, that handler will catch any exception no matter what the type of the throw exception is. This can be used as a default handler that catches all exceptions not caught by other handlers if it is specified at last:

```
try {
  // code here
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

In this case the last handler would catch any exception thrown with any parameter that is neither an int nor a char.

After an exception has been handled the program execution resumes after the try-catch block, not after the throw statement!.

It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression throw; with no arguments. For example:

```
try {
  try {
      // code here
  }
  catch (int n) {
      throw;
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

Exception specifications

When declaring a function we can limit the exception type it might directly or indirectly throw by appending a throw suffix to the function declaration:

```
float myfunction (char param) throw (int);
```

This declares a function called myfunction which takes one argument of type char and returns an element of type float. The only exception that this function might throw is an exception of type int. If it throws an exception with a different type, either directly or indirectly, it cannot be caught by a regular int-type handler.

If this throw specifier is left empty with no type, this means the function is not allowed to throw exceptions. Functions with no throw specifier (regular functions) are allowed to throw exceptions with any type:

```
1 int myfunction (int param) throw(); // no exceptions allowed
2 int myfunction (int param);         // all exceptions allowed
```

Standard exceptions

The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called exception and is defined in the <exception> header file under the namespace std. This class has the usual default and copy constructors, operators and destructors, plus an additional virtual member function called what that returns a null-terminated character sequence (char *) and that can be overwritten in derived classes to contain some sort of description of the exception.

```
// standard exceptions
#include <iostream>
#include <exception>
using namespace std;


class myexception: public exception
{
  virtual const char* what() const throw()
```

```
My exception happened.
```

```
 {
   return "My exception happened";
 }
} myex;


int main () {
 try
 {
   throw myex;
 }
 catch (exception& e)
 {
   cout << e.what() << endl;
 }
 return 0;
}
```

We have placed a handler that catches exception objects by reference (notice the ampersand & after the type), therefore this catches also classes derived from exception, like our myex object of class myexception.

All exceptions thrown by components of the C++ Standard library throw exceptions derived from this std::exception class. These are:

For example, if we use the operator new and the memory cannot be allocated, an exception of type bad_alloc is thrown:

```
try
{
  int * myarray= new int[1000];
}
catch (bad_alloc&)
{
```

```
  cout << "Error allocating memory." << endl;
}
```

It is recommended to include all dynamic memory allocations within a try block that catches this type of exception to perform a clean action instead of an abnormal program termination, which is what happens when this type of exception is thrown and not caught. If you want to force a bad_alloc exception to see it in action, you can try to allocate a huge array; On my system, trying to allocate 1 billion ints threw a bad_alloc exception.

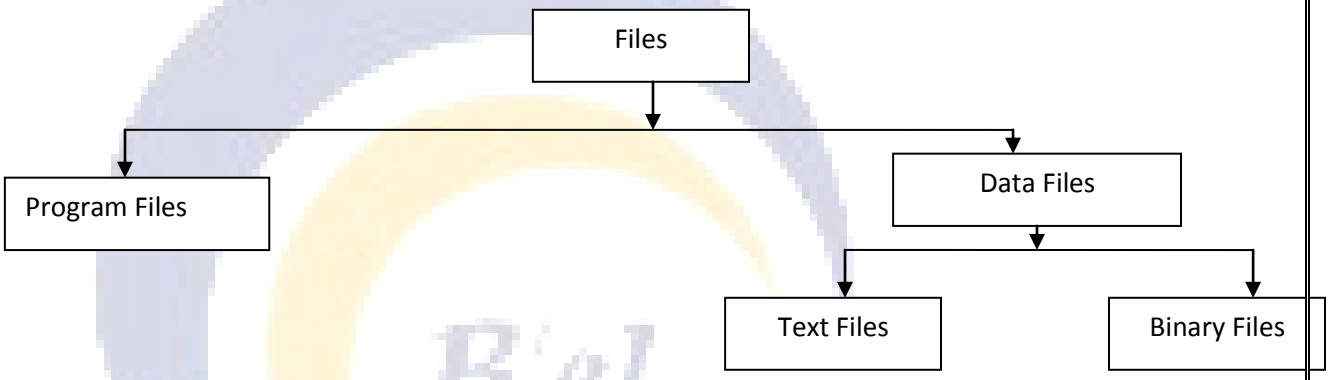
Because bad_alloc is derived from the standard base class exception, we can handle that same exception by catching references to the exception class:

```
// bad_alloc standard exception
#include <iostream>
#include <exception>
using namespace std;

int main () {
  try
  {
    int* myarray= new int[1000];
  }
  catch (exception& e)
  {
    cout << "Standard exception: " <<
e.what() << endl;
  }
  return 0;
}
```

Lesson 12

File Handling in C++

Files are required to save our data for future use, as Ram is not able to hold our data permanently.



The Language like C/C++ treat every thing as a file , these languages treat keyboard , mouse, printer, Hard disk , Floppy disk and all other hardware as a file.

The Basic operation on text/binary files are : Reading/writing ,reading and manipulation of data stored on these files. Both types of files needs to be open and close.

How to open File

Using member function Open( )	**Using Constructor**
Syntax Filestream object; Object.open(“filename”,mode); Example ifstream fin; fin.open(“abc.txt”)	Syntax Filestream object(“filename”,mode); Example ifstream fin(“abc.txt”);

NOTE: (a) Mode are optional and given at the end .

(a) Filename must follow the convention of 8.3 and it's extension can be anyone

How to close file

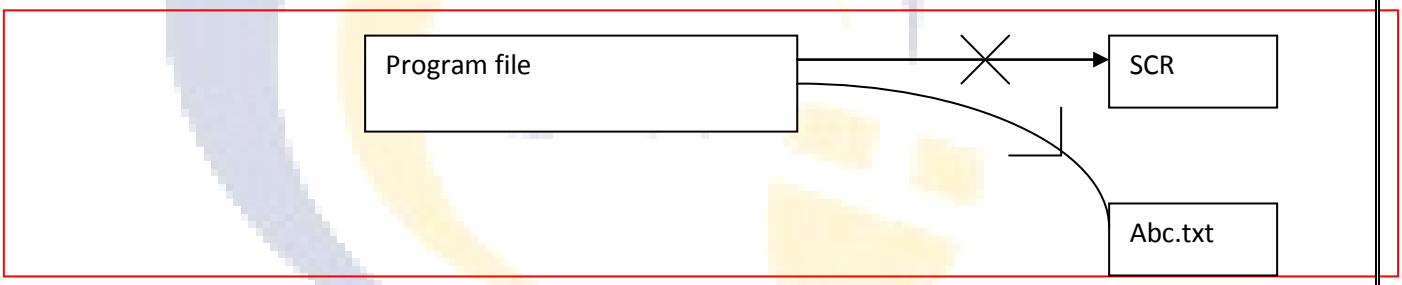
All types of files can be closed using **close( )** member function

Syntax

```
fileobject.close( );
```

Example

Objective : To insert some data on a text file



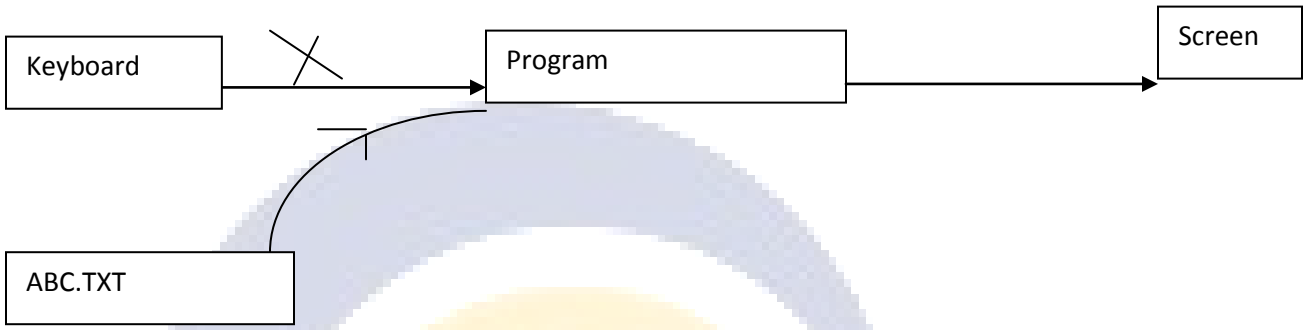
Program

Program	ABC.txt file contents

`#include<fstream>`
`using namespace std;`
`int main()`
`{`
`  ofstream fout;`
`  fout.open("abc.txt");`
`  fout<<"This is my first program in file handling";`
`  fout<<"\n Hello again";`
	`This is my first program in file handling`
	`Hello again`

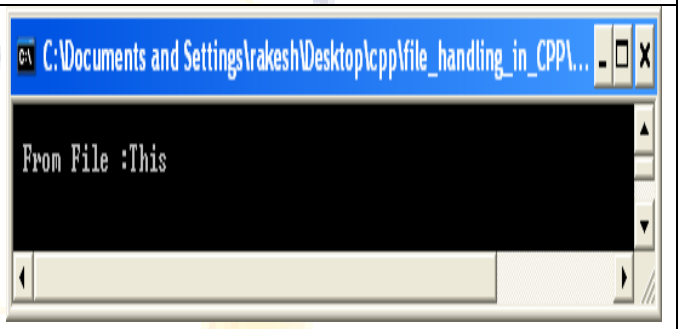
```
    fout.close();
    return 0;
}
```

Reading data from a Text File




```
#include<fstream>
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
    ifstream fin;
    char str[80];
    fin.open("abc.txt");
    fin>>str;     // read only
first //string from file
    cout<<"\n From File :"<<str;
// as         //spaces is treated
as termination point
    getch();
    return 0;
}
```

NOTE : To overcome this problem use
fin.getline(str,79);



Detecting END OF FILE

Using **EOF( ) member function**	Using **filestream object**

Syntax

```
Filestream_object.eof( );
```

Example

```
#include<iostream>
#include<fstream>
#include<conio.h>
using namespace std;
int main()
{
    char ch;
    ifstream fin;
    fin.open("abc.txt");
    while(!fin.eof())    // using eof()
                         // function
     {
       fin.get(ch);
       cout<<ch;
     }
   fin.close();
   getch();
   return 0;
}
```

Example

```
// detectting end of file
#include<iostream>
#include<fstream>
#include<conio.h>
using namespace std;
int main()
{
    char ch;
    ifstream fin;
    fin.open("abc.txt");
    while(fin) // file object
     {
       fin.get(ch);
       cout<<ch;
     }
   fin.close();
   getch();
   return 0;
}
```

Example **: To read the contents of a text file and display them on the screen**.

Program (using **getline** member function)

```
#include<fstream>
#include<conio.h>
#include<iostream>
using namespace std;
int main()
{
   char str[100];
   ifstream fin;
   fin.open("c:\\abc.txt");
   while(!fin.eof())
    {
        fin.getline(str,99);
        cout<<str;
        }
   fin.close();
   getch();
   return 0;
}
```

Program (using **get( )** member function)

```
#include<fstream>
#include<conio.h>
#include<iostream>
using namespace std;
int main()
{
    char ch;
    ifstream fin;
    fin.open("file6.cpp");
    while(!fin.eof())
     {
        fin.get(ch);
        cout<<ch;
        }
    fin.close();
    getch();
    return 0;
}
```

Writing/Reading Data in Binary Format

To write and read data in binary format two member functions are available in C++. They are read( ) and write( ).

Syntax for Write( ) member function

Fileobject.write((char *)&object, sizeof(object));

Syntax for read( ) member function

Fileobject.read((char *)&object, sizeof(object));

Example of write ( ) member function

Program	Output

```
#include<fstream>
#include<iostream>
using namespace std;
struct student
 {
       int roll ;
       char name[30];
       char address[60];
       };
int main()
 {
     student s;
     ofstream fout;
     fout.open("student.dat");
     cout<<"\n Enter Roll Number :";
     cin>>s.roll;
     cout<<"\n Enter Name :";
     cin>>s.name;
     cout<<"\n Enter address :";
     cin>>s.address;
     fout.write((char *)&s,sizeof(student));
     fout.close();
     return 0;
  }
```

To Read data from a binary File using read( ) member function

Program using **read( )** member function	output

```
#include<fstream>
#include<iostream>
#include<conio.h>
using namespace std;
struct student
```

```
 {
       int roll ;
       char name[30];
       char address[60];
       };
int main()
 {
     student s;
     ifstream fin;
     fin.open("student.dat");
     fin.read((char *)&s,sizeof(student));
     cout<<"\n Roll Number :"<<s.roll;
     cout<<"\n Name        :"<<s.name;
     cout<<"\n Address     :"<<s.address;
     fin.close();
     getch();
     return 0;
     }
```

Writing Class object in a file

```
#include<fstream>
#include<iostream>
using namespace std;
class student
 {
       int roll ;
       char name[30];
       char address[60];
     public:
           void read_data( );   // member function prototype
           void write_data( );  // member function prototype

     };
void student::read_data( )      // member function defintion
 {
     cout<<"\n Enter Roll :";
     cin>>roll;
     cout<<"\n Student name :";
     cin>>name;
     cout<<"\n Enter Address :";
     cin>>address;
}
void student:: write_data()
 {
     cout<<"\n Roll :"<<roll;
     cout<<"\n Name :"<<name;
     cout<<"\n Address :"<<address;
  }
int main()
```

```
 {
     student s;
     ofstream fout;
     fout.open("student.dat");
     s.read_data();  // member function call to get data from KBD
     fout.write((char *)&s,sizeof(student)); // write object in file
     fout.close();
     return 0;
     }
```

Reading Class object from a binary file

```
#include<fstream>
#include<iostream>
#include<conio.h>
using namespace std;
class student
 {
       int roll ;
       char name[30];
       char address[60];
     public:
           void read_data( );   // member function prototype
           void write_data( );  // member function prototype

       };
void student::read_data( )      // member function defintion
 {
     cout<<"\n Enter Roll :";
     cin>>roll;
     cout<<"\n Student name :";
     cin>>name;
     cout<<"\n Enter Address :";
     cin>>address;
}
void student:: write_data()
 {
     cout<<"\n Roll :"<<roll;
     cout<<"\n Name :"<<name;
     cout<<"\n Address :"<<address;
  }
int main()
 {
     student s;
     ifstream fin;
     fin.open("student.dat");
     fin.read((char *)&s,sizeof(student));
     s.write_data();
     fin.close();
     getch();
```

```
return 0;
}
```

some other very important member function

Member function name	Explanation

seekg( )	Used to move **reading** pointer forward and backward
	Syntax
	fileobject.seekg(no_of_bytes,mode);
	Example:
	(a) fout.seekg(50,**ios::cur**); // move 50 bytes forward from current position
	(b) fout.seekg(50,**ios::beg**); // move 50 bytes forward from current beginning
	(c) fout.seekg(50,**ios::end**); // move 50 bytes forward from end .

seekp( )	Used to move **writing** pointer forward and backward
	Syntax
	fileobject.seekp(no_of_bytes,mode);
	Example:
	(a) fout.seekp(50,**ios::cur**); // move 50 bytes forward from current position
	(b) fout.seekp(50,**ios::beg**); // move 50 bytes forward from current beginning
	(c) fout.seekp(50,**ios::end**); // move 50 bytes forward from end .

tellp( )	It return the distance of **writing** pointer from the beginning in bytes Syntax Fileobject.tellp( ); Example: long n = fout.tellp( );
tellg( )	It return the distance of **reading** pointer from the beginning in bytes Syntax Fileobject.tellg( ); Example: long n = fout.tellg( );

Files MODES

File mode	**Explanation**
ios::in	Input mode – Default mode with **ifstream** and files can be read only
ios::out	Output mode- Default with **ofstream** and files can be write only

ios::binary	Open file as binary
ios::app	Preserve previous contents and write data at the end (move forward only)
ios::ate	Preserve previous contents and write data at the end.(can move forward and backward)
ios::nodelete	Do not delete existing file
ios::noreplace	Do not replace file
ios::nocreate	Do not create file

NOTE : To add more than one mode in a file stream use bitwise OR (|) operator

Example :

Difference and Definition_____

Text Files	Binary Files
In these types of files all the data is firstly converted into their equivalent char and then it is stored in the files.	In these types of files all the data is stored in the binary format as it is stored by the operating system. so no conversion takes place. Hence the processing speed is much more than text files.

get( ) member function	**getline()** function

Get() function is used to read a single char from the input stream in text file
Syntax
fileobject.get(char);
Example:
Fin.get(ch); //fin is file stream.
	Getline() function is used to read a string from the input stream in text file.
	Syntax
	fileobject.getline (string, no_of_char, delimiter);
	Example
	fin.getline(str,80); // fin is file stream.
	NOTE: Delimiter is optional

Program to Explain the different operation for Project

```
#include<iostream>
#include<fstream>
#include<conio.h>
using namespace std;
class student
   {
         int admno;
         char name[30];
         char address[60];
       public:
         void read_data()
           {
              cout<<"\n Enter Admission No :";
              cin>>admno;
              fflush(stdin);
              cout<<"\n Enter Name    :";
              cin.getline(name,29);
              fflush(stdin);
              cout<<"\n Enter Address  :";
              cin.getline(address,59);
            }
         void write_data()
           {
              cout<<"\n\n Admission No   :"<<admno;
              cout<<"\n Name          :"<<name;
              cout<<"\n Address       :"<<address;
           }
        int get_admno()
          {
              return admno;
          }
};

void write_to_file(void)
 {
     student s;
     ofstream fout;
     fout.open("student.dat",ios::app);
     s.read_data();
     fout.write((char *)&s,sizeof(student));
     fout.close();
```

```
     return;
     }
 void read_from_file()
  {
      student s;
      ifstream fin;
      fin.open("student.dat");
      while(fin.read((char *)&s,sizeof(student)))
         s.write_data();
      fin.close();
      return;
      }

// function to modify student information

void modify_record(void)
 {
     int temp_admno;
     student s;
     ifstream fin;
     ofstream fout;
     fin.open("student.dat");
     fout.open("temp.dat");
     system("cls");     // header file stdlib.h
     cout<<"\n Enter admission No to Modify  :";
     cin>>temp_admno;
     while(fin.read((char *)&s,sizeof(student)))
       {
          if (temp_admno==s.get_admno())
            {
               s.read_data();
            }
           fout.write((char *)&s,sizeof(student));
       }
       fin.close();
       fout.close();
       remove("student.dat");
       rename("temp.dat","student.dat");
       return;
 }

void modify_alternate_method()
{
  student s;
  int temp_admno;
  fstream file;
  file.open("student.dat",ios::in|ios::out|ios::ate|ios::binary);
  cout<<"\n Enter admno to modify :";
  cin>>temp_admno;
  file.seekg(0);  // one method to reach at begining

//  long n = file.tellg();     // find out total no of bytes
 // file.seekg((-1)*n,ios::end); // move backward total no of bytes from end
```

```
  while(file.read((char*)&s,sizeof(student)))
    {
       if(temp_admno == s.get_admno())
         {
            s.read_data();
            int n = -1*sizeof(student);
            file.seekp(n,ios::cur);
            file.write((char *)&s,sizeof(student));
         }
     }
  file.close();
  return;
 }

void delete_record(void)
{
     int temp_admno;
     student s;
     ifstream fin;
     ofstream fout;
     fin.open("student.dat");
     fout.open("temp.dat");
     system("cls");
     cout<<"\n Enter admission No to Delete  :";
     cin>>temp_admno;
     while(fin.read((char *)&s,sizeof(student)))
       {
          if (temp_admno!=s.get_admno())
                    fout.write((char *)&s,sizeof(student));
       }
       fin.close();
       fout.close();
       remove("student.dat");              //  stdio.h
       rename("temp.dat","student.dat");  // stdio.h
       return;
 }

void search_record()
 {
    int found=0;
    student s;
    int temp_admno;
    ifstream fin("student.dat");
    cout<<"\n Enter Admno to search :";
    cin>>temp_admno;
    while(fin.read((char*)&s,sizeof(student)))
      {
         if(temp_admno==s.get_admno())
           {
              found=1;
              s.write_data();
           }
```

```
        }
    fin.close();
    if(found ==0)
             cout<<"\n Admission No. "<<temp_admno<<" does not exist ";
    getch();
    return;
}


void count_record(void)
{
    int count=0;
    student s;
    int temp_admno;
    ifstream fin("student.dat");
    while(fin.read((char*)&s,sizeof(student)))
                  count++;
    fin.close();
    cout<<"\n Total Record :"<<count;
    getch();
    return ;
 }


int main()
 {
       int choice;
       do
         {
              system("cls");                                  // stdlib.h
              cout<<"\n\n\t\t\t           MAIN MENU ";
              cout<<"\n\t\t\t\t1.      Add Student ";
              cout<<"\n\t\t\t\t2.      Show Student";
              cout<<"\n\t\t\t\t3.      Modify Record";
              cout<<"\n\t\t\t\t4.      Modify Record (Alternate Method)";
              cout<<"\n\t\t\t\t5.      Delete Record";
              cout<<"\n\t\t\t\t6.      Count Record";
              cout<<"\n\t\t\t\t7.      Search Record";
              cout<<"\n\t\t\t\t8.      Exit";
              cout<<"\n\n\t\t\t\t\t    Enter your choice :";
              cin>>choice;
              switch(choice)
               {
                   case 1:        system("cls");
                                  write_to_file();
                                  break;
                   case 2:
                                  read_from_file();
                                  getch();
                                  break;
                   case 3:
                                  modify_record();
                                  break;
```

```
                    case 4:
                                modify_alternate_method();
                                break;
                    case 5:
                                delete_record();
                                break;
                    case 6:     count_record();
                                break;
                    case 7:     search_record();
                                break;
                    case 8:
                                break;

                    default:
                                cout<<"\n Wrong choice.... Try again";
                                getch();
                  }
             }while(choice!=8);
    return 0;
}
```