C++

Variable

- variables are used to store various information.
- Variables are nothing but reserved memory locations to store values.
- when a variable is created we reserve some space in memory.

Rules to create a variable:

- A variable is a user defined name it can be defined as follows
  - It has to start with an alphabet.
  - It can be followed by any number of alphabet or numbers
  - Only underscore _ character is allowed while defining a variable
  - Special characters are not allowed
  - Whitespaces cannot be used
  - They are case sensitive : car, Car, CAR, CaR, cAr, cAR… all are different variables.

Syntax for defining a variable:

Declaration

Datatype variablename;

Initialization

Datatype variablename = value;

Examples for variable

1. int count;
2. double b1=123.4567;
3. Char c;
4. int arr[10];
5. float ary[]={12.3,23.4,34.5,45.6,56.7}

DataTypes

- Information can be stored in various data types like character, wide character, integer, floating point, double floating point, boolean etc.
- Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

Primitive Built-in Types

C++ offers the programmer a rich assortment of built-in as well as user defined

data types. Following table lists down seven basic C++ data types –

Type	Keyword
Boolean	Bool
Character	Char
Integer	Int
Floating point	``
Double floating point	Double
Valueless	Void
Wide character	wchar_t

Several of the basic types can be modified using one or more of these type modifiers –

- signed
- unsigned
- short
- long

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255

unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	Range	0 to 65,535
signed short int	Range	-32768 to 32767
long int	4bytes	-2,147,483,648 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	2 or 4 bytes	1 wide character

- The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

Following is the example, which will produce correct size of various data types on your computer.

```
#include<iostream.h>
#include<conio.h>
int main(){
cout<<"Size of char : "<<sizeof(char)<<endl;
cout<<"Size of  int : "<<sizeof(int)<<endl;
cout<<"Size of shot int : "<<sizeof(short int)<<endl;
cout<<"Size of long int: "<<sizeof(long int)<<endl;
cout<<"Size of float : "<<sizeof(float)<<endl;
cout<<"Size of double : "<<sizeof(double)<<endl;
cout<<"Size of wchar_t : "<<sizeof(wchar_t)<<endl;
return 0;
}
```

This example uses **endl**, which inserts a new-line character after every line and << operator is being used to pass multiple values out to the screen. We are also using **sizeof()** operator to get size of various data types.

When the above code is compiled and executed, it produces the following result which can vary from machine to machine −

```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8
Size of wchar_t : 4
```

Typedef

You can create a new name for an existing type using **typedef**. Following is the simple syntax to define a new type using typedef −

typedef type newname;

For example typedef Declarations

the following tells the compiler that feet is another name for int −

```
typedef int feet;
```

Now, the following declaration is perfectly legal and creates an integer variable called distance −

```
feet distance;
```

Enumerated Types

An enumerated type declares an optional type name and a set of zero or

more identifiers that can be used as values of the type.

Each enumerator is a constant whose type is the enumeration.

Creating an enumeration requires the use of the keyword **enum**. The general form of an enumeration type is −

enum enum-name { list of names } var-list;
The enum-name is the enumeration's type name. The list of names is comma separated.

For example, the following code defines an enumeration of colors called colors and the variable c of type color. Finally, c is assigned the value "blue".

```
enum color { red, green, blue } c;
c = blue;
```

By default, the value of the first name is 0, the second name has the value 1, and the third has the value 2, and so on. But you can give a name, a specific value by adding an initializer. For example, in the following enumeration, **green** will have the value 5.

```
enum color { red, green = 5, blue };
```

Here, **blue** will have a value of 6 because each name will be one greater than the one that precedes it.

Tokens

A token is the smallest element of a C++ program that is meaningful to the compiler. The C++ parser recognizes these kinds of tokens: identifiers, keywords, literals, operators, punctuators, and other separators. A stream of these tokens makes up a translation unit.

Tokens are usually separated by *white space*. White space can be one or more:

- Blanks
- Horizontal or vertical tabs
- New lines
- Form feeds
- Comments

The parser recognizes keywords, identifiers, literals, operators, and punctuators.

The following are the tokens defined and used in c++

- Keywords, Identifiers, Numeric, Boolean and Pointer Literals, String and Character Literals, User-Defined Literals, C++ Built-in Operators, Precedence and Associativity, and Punctuators. White space is ignored, except as required to separate tokens.

- Preprocessing tokens are used in the preprocessing phases to generate the token stream passed to the compiler.

- The preprocessing token categories are header names, identifiers, preprocessing numbers, character literals, string literals, preprocessing operators and punctuators, and single non-white-space characters that do not match one of the other categories.

- Character and string literals can be user-defined literals. Preprocessing tokens can be separated by white space or comments.

The parser separates tokens from the input stream by creating the longest token possible using the input characters in a left-to-right scan. Consider this code fragment:

int a, i,j; a=i+++j;

the expression will be evaluated taking the longest token possible. Thus the expression is evaluated as

a = (i++) + j;

Identifiers

A C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item.

An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9).

C++ does not allow punctuation characters such as @, $, and % within identifiers. C++ is a case-sensitive programming language.

Thus, **Manpower** and **manpower** are two different identifiers in C++.

Here are some examples of acceptable identifiers –

mohd zara abc move_name a_123
myname50 _temp j a23b9 retVal

Keywords

Keywords are reserved words that are specially used to perform certain specific job.

These reserved words may not be used as constant or variable or any other identifier names.

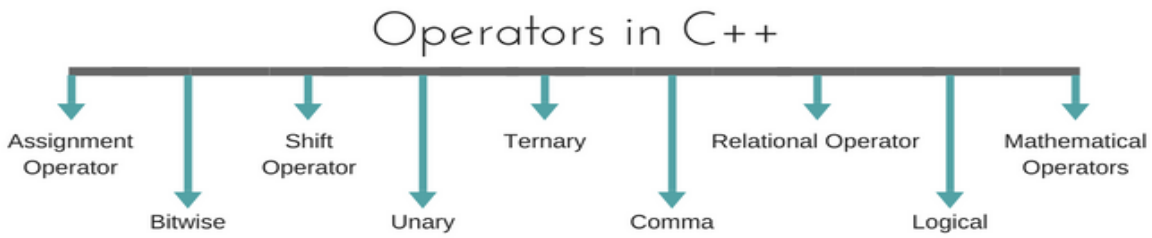
The following list shows the reserved words in C++.

asm	else	new	this
auto	enum	operator	throw

bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

Operators in C++

Operators are special type of functions, that takes one or more arguments and produces a new value. For example : addition (+), substraction (-), multiplication (*) etc, are all operators. Operators are used to perform various operations on variables and constants.



Types of operators

1. Assignment Operator
2. Mathematical Operators
3. Relational Operators
4. Logical Operators
5. Bitwise Operators
6. Shift Operators
7. Unary Operators
8. Ternary Operator
9. Comma Operator

Assignment Operator (=)

Operates '=' is used for assignment, it takes the right-hand side (called rvalue) and copy it into the left-hand side (called lvalue). Assignment operator is the only operator which can be overloaded but cannot be inherited.

Mathematical Operators

There are operators used to perform basic mathematical operations. Addition (+) , subtraction (-) , diversion (/) multiplication (*) and modulus (%) are the basic mathematical operators. Modulus operator cannot be used with floating-point numbers.

C++ and C also use a shorthand notation to perform an operation and assignment at same type. *Example*,

```
int x=10;
x += 4 // will add 4 to 10, and hence assign 14 to X.
```

```
x -= 5 // will subtract 5 from 10 and assign 5 to x.
```

Relational Operators

These operators establish a relationship between operands. The relational operators are

Less than <, less that and equal to <=

Greater than >, greater than and equal to >=, equivalent == and not equivalent !=.

Example:

```
int x = 10;  //assignment operator
x=5;        // again assignment operator
if(x == 5)   // here we have used equivalent relational operator, for comparison
{
   cout <<"Successfully compared";
}
```

Logical operators

The logical operators are AND (&&) and OR (||). They are used to combine two different expressions together.

If two statement are connected using AND operator, the validity of both statements will be considered, but if they are connected using OR operator, then either one of them must be valid. These operators are mostly used in loops (especially while loop) and in Decision making.

Bitwise operator

These are used to change individual bits into a number. They work with only integral data type like char, int and long and not with floating point values.

Bitwise AND operator &
Bitwise OR|
Bitwise XOR operator ^
Bitwise NOT operator ~

Shift Operators

Shift operators are used to shift Bits of any variable. It is of three types

Left Shift operatpr <<
Right shift operator >>
Unsigned Right shift operator >>>

Unary operators

These are the operators which work ononly one operand. These are many unary operators, but increment ++ and decrement – operators are most used.
Other unary operators:
Address of &, dereference *, new and delete, bitwise not ~,logical not !, unary minus – and unary plus +.

Ternary operator

The ternary if-else ?: is an operator which has three operands.
Example:
int a=10;
a>5? Cout << “true” ; cout <<”false”;

Comma operator

This is used to separate variable names and to separate expressions. In case of expressions, the value of last expression is produced and used.

Example:

```
int a,b,c;

a=b++, c++; // a=c++ will be done
```

Arrays

C++ provides a data structure, **the array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays

To declare an array in C++, the programmer specifies the type of the elements and

the number of elements required by an array as follows −

type arrayName [arraySize];

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement −

double balance[10];

Initializing Arrays

You can initialize C++ array elements either one by one or using a single statement as follows −

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array −

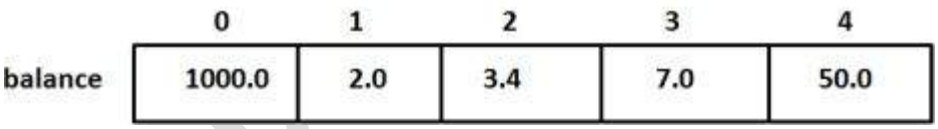
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};

You will create exactly the same array as you did in the previous example.

balance[4] = 50.0;

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representaion of the same array we discussed above −



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays −

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

#include <iomanip>
using std::setw;
```

```
int main () {

  int n[ 10 ]; // n is an array of 10 integers

  // initialize elements of array n to 0
  for ( int i = 0; i < 10; i++ ) {
    n[ i ] = i + 100; // set element at location i to i + 100
  }
  cout << "Element" << setw( 13 ) << "Value" << endl;

  // output each array element's value
  for ( int j = 0; j < 10; j++ ) {
    cout << setw( 7 )<< j << setw( 13 ) << n[ j ] << endl;
  }

  return 0;
}
```

This program makes use of **setw()** function to format the output. When the above code is compiled and executed, it produces the following result –

```
Element      Value
    0        100
    1        101
    2        102
    3        103
    4        104
    5        105
    6        106
    7        107
    8        108
    9        109
```

C++ allows to create and access the following

1. C++ supports multidimensional arrays. The simplest form of the multidimensional array is two-dimensional array.
   Syntax
   Datatype variablename[int][int];
   Example
   Float farr[5][5]; //5*5 matrix that holds floating point values
2. Pointers can be generated to the first element of an array by simply specifying the array name, without any index.
3. C++ allows a function to return an array.
4. A pointer to an array can be passed to a function by specifying array name without an index.

I/O streams
CIN and COUT functions
the c++ standard libraries provide an extensive set of input/output capabilities.

C++ i/o occurs in streams, which are sequences of bytes. If bytes flow from a device like keyboard, a disk drive or a network connection etc., to main memory, this is called input operation.

If bytes flow from main memory to a device like a display screen, a printer, a disk drive or a network connection etc., this is called output operation.

I/O Library header files

<iostream>

This header file defines the cin, cout, cerr and clog objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream respectively.

The Standard Output Stream (cout)

The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout** is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

Example:

```
#include <iostream>
void main()
{
  char str[]=”hello C++”;
  cout <<”value of str is :    “<<str<<endl;
}
```

The Standard Input Stream (cin)

The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

Example:

```
#include <iostream>

int main()

{

char name[50];

ccout << “ please enter your name :  “;

cin >> name;
```

```
cout <<" your name is   : " << name <<endl;
}
```

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;

Forexample:

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

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

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

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

Iteration structures (loops)

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

The while loop

Its format is:

while (expression) statement

and its functionality is simply to repeat statement while the condition set in expression is true.
For example, we are going to make a program to countdown using a while-loop:

```
// custom countdown using while

#include <iostream>
using namespace std;

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

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

  cout << "FIRE!\n";
  return 0;
}
```

```
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

When the program starts the user is prompted to insert a starting number for the countdown. Then the while loop begins, if the value entered by the user fulfills the condition n>0 (that n is greater than zero) the block that follows the condition will be executed and repeated while the condition (n>0) remains being true.

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

1. User assigns a value to n
2. The while condition is checked (n>0). At this point there are two 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>
using namespace std;

int main ()
{
```

```
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
```

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

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

It works in the following way:

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

Here is an example of countdown using a for loop:

```
// countdown using a for loop
#include <iostream>
using namespace std;
int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "FIRE!\n";
  return 0;
}
```

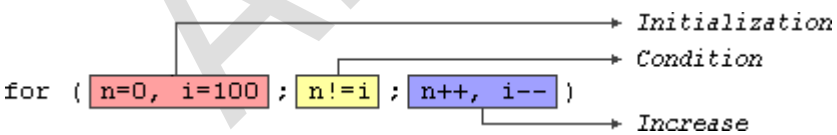
```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: for (;n<10;) if we wanted to specify no initialization and no increase; or for (;n<10;n++) if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before).

Optionally, using the comma operator (,) we can specify more than one expression in any of the fields included in a for loop, like in initialization, for example. The comma operator (,) is an expression separator, it serves to separate more than one expression where only one is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:

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

This loop will execute for 50 times if neither n or i are modified within the loop:



n starts with a value of 0, and i with 100, the condition is n!=i (that n is not equal to i). Because n is increased by one and i decreased by one, the loop's condition will become false after the 50th loop, when both n and i will be equal to 50.

Jump Statement

The break statement

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

```
// break loop example

#include <iostream>
using namespace std;

int main ()
{
  int n;
  for (n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
  return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!
```

The continue statement

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

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

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!
```

```
// goto loop example

#include <iostream>
using namespace std;

int main ()
{
  int n=10;
  loop:
  cout << n << ", ";
  n--;
  if (n>0) goto loop;
  cout << "FIRE!\n";
  return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

```
 }
 cout << "FIRE!\n";
 return 0;
}
```

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:

The exit function

exit is a function defined in the cstdlib 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 it 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.

Expressions

- An expression is a sequence of *operators* and their *operands* that specifies a computation.
- An expression is a combination of variables constants and operators written according to the syntax of C++ language.
- In C++ every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable.
- Some examples of C++ expressions are shown in the table given below.

Algebraic Expression	C++ Expression
a x b – c	a * b – c
(m + n) (x + y)	(m + n) * (x + y)
(ab / c)	a * b / c
$3x^2$ +2x + 1	3*x*x+2*x+1
(x / y) + c	x / y + c

Expression Evaluation

- Expressions are evaluated using an assignment statement of the form
  Variable = expression;
- Variable is any valid C++ variable name.
- When the statement is encountered, the expression is evaluated first and then replaces the previous value of the variable on the left hand side.
- All variables used in the expression must be assigned values before the evaluation is attempted.

C++ UNIT II

Evolution of C++

- ✔ The C++ language is an object-oriented programming language & is a combination of both low-level & high-level language – a Middle-Level Language.
- ✔ The programming language was created, designed & developed by a Danish Computer Scientist – Bjarne Stroustrup at Bell Telephone Laboratories (now known as Nokia Bell Labs) in Murray Hill, New Jersey.
- ✔ As he wanted a flexible and a dynamic language which was similar to C with all its features, but with additionality of active type checking, basic inheritance, default functioning argument, classes, inlining, etc. and hence C with Classes (C++) was launched.
- ✔ C++ was initially known as "C with classes" and was renamed C++ in 1983.
- ✔ ++ is shorthand for adding one to variety in programming.
- ✔ Therefore C++ roughly means that "one higher than C."

Object-Oriented Programming Paradigm

- OOP removes some of the flaws encountered in the procedural approach.
- OOP treats data as a critical element in the program development and does not allow it to flow freely around the system.
- It ties data more closely to the functions that operate on it.
- OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects.

Some of the striking features of object-oriented programming are:

• Emphasis is on data rather than procedure.

• Programs are divided into what are known as objects.

• Data structures are designed such that they characterize the objects.

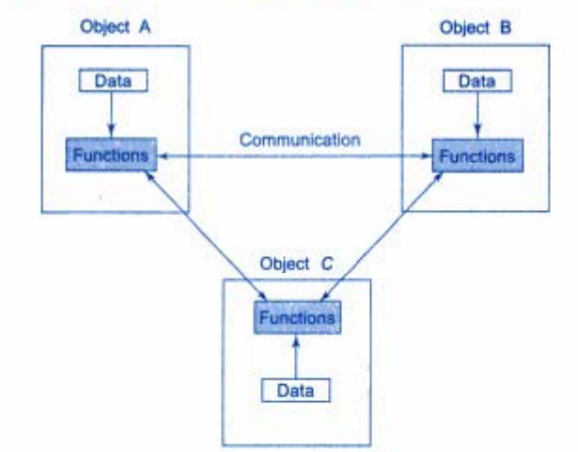
• Functions that operate on the data of an object are tied together in the data structure.

• Data is hidden and cannot be accessed by external functions.

• Objects may communicate with each other through functions.

• New data and functions can be easily added whenever necessary.

• Follows bottom-up approach in program design.



Basic Concepts of Object-Oriented Programming/Key Concept of OOPs

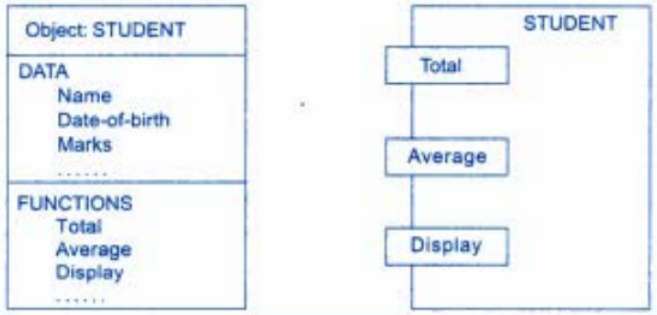
- Objects
- Classes
- Data abstraction
- encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

Objects

- Objects are the basic run· time entities in an object · oriented system.
- They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.

- They may also represent user· defined data such as vectors, time and lists.
- Programming problem is analyzed in terms of objects and the nature of communication between them.
- Objects take up space in the memory and have an associated address like a record in Pascal,

  or a structure in C.
- When a program is executed, the objects interact by sending messages to one another.



Classes

- Objects contain data and code to manipulate the data.
- The entire set of data and code of an object can be made a user-defined data type with the help of class.
- Objects are variables of the type class.
- Once a class has been defined, any number of objects can be created belonging to that class.
- Each object is associated with the data of type class with which they are created. A class is thus a collection of objects of similar type.
- For example, mango, apple and orange are members of the class fruit.
- Classes are user-defined data types and behave like the built-in types of a programming language.

- The syntax used to create an object is no different than the syntax used to create an integer object in C.
- If fruit bas been defined as a class, then the statement

  fruit mango;

  will create an object mango belonging to the class fruit .

Data Abstraction and Encapsulation

- The wrapping up of data and functions into a single unit (called c1ass) is known as Encapsulation.
- Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it.
- These functions provide the interface between the object's data and the program.
- This insulation of the data from direct access by the program is called data hiding or information hiding.

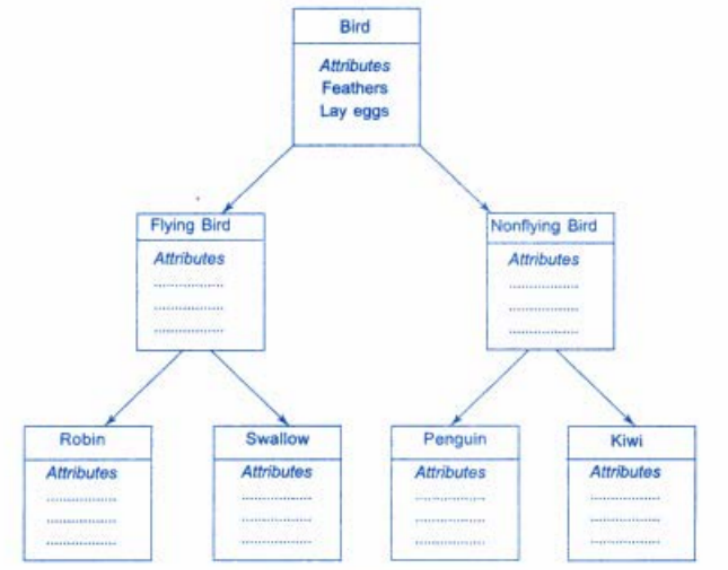
Abstraction

- It refers to the act of representing essential features without including the background details or explanations.
- Classes use the concept of abstraction and are defined as a list of abstract attribute such as size, weight, cost and functions to operate on the attributes.
- They encapsulate all the essential properties of the objects that are to be created.
- The attributes are sometimes called data numbers because they hold information.
- The functions that operate on these data are sometimes called methods or member functions.
- Since the classes have the concept of data abstraction, they are known as Abstract Data Types (ADT).

Inheritance

- Inheritance is the process by which objects of one class acquire the properties of objects of another class.

- It supports the concept of hierarchical classification.
- For example, the bird 'robin' is a part of the class 'flying bird' which is again a part of the class 'bird'. The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated below.
- In OOP, the concept of inheritance provides the idea of reusability.
- This means that additional features can be added to an existing class without modifying it. This is possible by deriving a new class from the existing on
- The new class will have the combined features of both the classes.

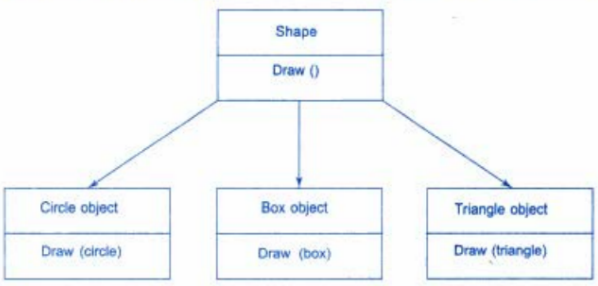


Polymorphism

- Polymorphism is another important OOP concept.
- Polymorphism is a Greek term, means the ability to take more than one form. An operation may exhibit different behavior in different instances.
- The behavior depends upon the types of data used in the operation.
- For example, consider the operation of addition, for two numbers, the operation will generate a sum. If the operands are strings, then the operation would

produce a third string by concatenation.

- The process of making an operator to exhibit different behavior in different instances is known as operator overloading.
- Fig. below illustrates a single function name can be used to handle different number and different types of arguments.
- This is something similar to a particular word having several different meanings depending on the context.
- Using a single function name to perform different types or tasks is known as; function overloading.
- Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface.
- This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.



Dynamic Binding

- Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time.
- It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.
- At run-time, the code matching the object under current reference will be called.

Message Passing

- An object Oriented program consists of a set of objects that communicate with

each other.

- process of programing in an object-oriented language, therefore, involves the following basic steps:

    1. Creating classes that define objects and their behaviour,
    2. Creating objects from class definitions and
    3. Establishing communication among objects.

- Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another.
- A message for an object is a request for execution of a procedure and therefore invokes a function (procedure) in the receiving object that generates the desired result.
- Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent. Example:

- Objects have a life cycle. They can be created and destroyed.
- Communication with an object is feasible as long as it is alive.

Benefits of OOP/Advantages of OOP

- OOP offers several benefits to both the program designer and the user.
- Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:
    - Through inheritance, we can eliminate redundant code and extend the use of existing classes.

- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- A message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

Applications of OOP/Usage of OOP

OOP has become one of the programming buzzwords today. There appears to be a great deal of excitement and interest among software engineers in using OOP.

Applications of OOP are beginning to gain importance in many areas.

Hundreds of windowing systems have more complex and contain many more objects with complicated attributes and methods.

OOP is useful in these types of applications because it can simplify a complex

problem.

The promising areas for application of OOP include:

- Real-time systems
- Simulation and modeling
- Object-oriented databases
- Hypertext, hypermedia and expertext
- Al and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

The richness of OOP environment has enabled the software industry to improve not only the quality of software systems but also its productivity. Object-oriented technology is certainly changing the way the software engineers think, analyze, design and implement systems.

Advantages of C++

- C++ is a highly portable language and is often the language of selection for multi-device, multi-platform app development.
- C++ is an object-oriented programming language and includes concepts like classes, inheritance, polymorphism, data abstraction, and encapsulation which allow code reusability and makes programs very maintainable.
- C++ use multi-paradigm programming. The Paradigm means the style of programming .paradigm concerned about logics, structure, and procedure of the program. C++ is multi-paradigm means it follows three paradigm Generic, Imperative, Object Oriented.
- It is useful for the low-level programming language and very efficient for general purpose.
- C++ gives the user complete control over memory management. This can be seen both as an advantage and a disadvantage as this increases the responsibility of the user to manage memory rather

than it being managed by the Garbage collector.

- The wide range of applications – From GUI applications to 3D graphics for games to real-time mathematical simulations, C++ is everywhere.
- C++ has a huge community around it. Community size is important, because the larger a programming language community is, the more support you would be likely to get. C++ is the 6th most used and followed tag on StackOverflow and GitHub.
- C++ has a very big job market as it is used in various industries like finance, app development, game development, Virtual reality, etc.
- C++'s greatest strength is how scalable it could be, so apps that are very resource intensive are usually built with it. As a statically written language, C++ is usually more performant than the dynamically written languages because the code is type-checked before it is executed.
- Compatibility with C – C++ is compatible with C and virtually every valid C program is a valid C++ program.

Unit III

CLASSES

Class is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating instance of that class.

The variables inside class definition are called as data members and the functions are called member functions.

Properties of class

1. Class name must start with an alphabet followed by letters or digits, no special characters are allowed except underscore (_) .
2. Classes contain, data members and member functions, and the access of these data members and variable depends on the access specifiers.
3. Class's member functions can be defined inside the class definition or outside the class definition.
4. Class in C++ are similar to structures in C, the only difference being, class defaults to private access control, where as structure defaults to public.
5. All the features of OOPS, revolve around classes in C++. Inheritance, Encapsulation, Abstraction etc.
6. Objects of class holds separate copies of data members. We can create as many bjects of a class as we need.

Access Control in Classes

Access specifiers in C++ class defines the access control rules. C++ has 3 new keywords introduced, namely,

1. Public - The data members and member functions declared public can be accessed by other classes
2. Private - no one can access the class members declared private outside that class private

3. Protected - it makes class member inaccessible outside the class. But they can be accessed by any subclass of that class.

These access specifiers are used to set boundaries for availability of members of class be it data members or member functions. Access specifiers in the program, are followed by a colon.

Example of a class declaration and definition

```
class Abc
{
 int x;  //data member
//member function
Public:
void show()
{
   cout<<"\n example of a class in c++ ";
}
 void display(){} //empty function
};
```

Objects

Objects are instances of a class. Objects represent the combination of variables and functions of a class. Only through the objects the members of the class can be accessed.

Declaration of the Objects:

Objects can only be declared in the main function.

Any variable name can be declared as an object.

Objects access the data members and functions of the class using the DOT operator.

Example for Objects:

Class sample

```
{
        int a=5;  //class variable
Public:
        Void show()  //method or function
{
Cout <<"\n The value of a is : "<<a;
}
};
Void main()
{
  Sample ob1; // declaring objects for classes
Cout<<ob1.a;  //accessing variable of a class using objects.
Ob1.show(); //accessing method of a class using objects
}
```

Constructors

- A constructor is a kind of member function that initializes an instance of its class.
- When a constructor is invoked it allocates memory to all the data members and functions.
- A constructor has the same name as the class and no return value.
- A constructor can have any number of parameters and a class may have any number of overloaded constructors.
- Constructors should be declared as public.
- If you don't define any constructors, the compiler will generate a default constructor that takes no parameters.
- Constructors can be overrided.
- The constructor is invoked whenever an object of its associated class is created.
- Constructors cannot be virtual and cannot be derived in the sub class.
- The constructor makes an implicit call to the new operator when they are invoked.

Types of constructors

1. Default constructor

--Default constructor has no return type, no code defined in it, it does not return any value.

2. Parameterized constructors
    a. Single parameter

        --this type of parameter takes one parameter.

    b. Multiple parameters

        --this type takes two or more parameter as required.

3. Constructor with default parameter
    - The parameter declared in the constructor is also defined with a constant value.

4. Copy constructor

    ---A copy constructor is used to declare and initialize an object from another object.

    --- The process of initializing through a copy constructor is known as *copy initialization.*

Example:

```
Class Intmanipulate
{
 Public:
Intmanipulate(){}  //default constructor
Intmanipulate(int a) //single parameter constructor
{
   a=a+(a+10);
cout<<"\nSingle parameter constructor demo : "<<a<<endl;
}
Intmanipulate(int x, int y) //constructor with more than two parameter
{
  x = x*y;
cout<<"\n more than two parameter constructor demo : "<<x<<endl;
}
Intmanipulate(float pi=3.14, int z=100) //default parameter in constructor
```

```
{
  z= pi*z;
 cout<<"\nconstructor with default parameters demo : "<<z<<endl;
}

~Intmanipulate(){} //destructor function.
};

void main()
{
  Intmanipulate im, im101(50), im11(3,5);
  Intmanipulate im2=im; //copy constructor
}
```

Destructor

- A destructor is used to destroy the objects that have been created by a constructor.
- Like constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.

eg: ~ Intmanipulate( ) { }

- It is a good practice to declare destructors in a program since it releases memory space for further use.
- Whenever ***new*** is used to allocate memory in the constructor, we should use ***delete*** to free that memory.

Inheritance

- ✔ Inheritance is an object oriented technique used for code reuse.
- ✔ The technique of deriving a new class from an old one is called inheritance.
- ✔ The old class is referred to as base class and the new class is referred to as derived class or subclass.
- ✔ Inheritance concept allows programmers to define a class in terms of another class, which makes creating and maintaining application easier.
- ✔ When writing a new class, instead of writing new data member and member functions all over again, programmers can make a bonding of the new class with

the old one that the new class should inherit the members of the existing class.

- A class can get derived from one or more classes, which means it can inherit data and functions from multiple base classes.

Syntax

class derived-class : visibility-mode base-class

- Visibility mode takes either of the two types. Protected or public.

Base class and Derived class

- The existing class from which the derived class gets inherited is known as the base class. It acts as a parent for its child class and all its properties i.e. public and protected members get inherited to its derived class.
- A derived class can be defined by specifying its relationship with the base class in addition to its own details, i.e. members.

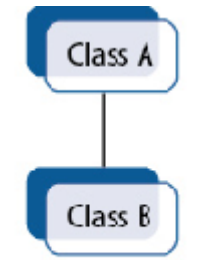
```
Syntax

Class base
{
        Protected:
        //data members
        Public:
        //member functions
};
Class derived:public base
{
        //members of the derived class.
};
```

Types of Inheritance

C++ offers five types of Inheritance. They are:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance (also known as Virtual Inheritance)

Single inheritance



Single Inheritance

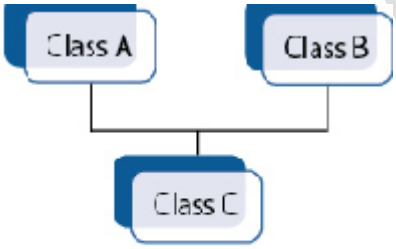
- This is the simplest form of inheritance.
- In single inheritance one base class is derived into a single derived class.
- All the members declared as public in the base class are accessible in the derived class.

Example:

```
class baseb
{
protected:
        int a;
 public :
        void get(){ cin>>a;}
        void cal(){ a=a*10;}
};
```

```
class derived : public baseb
{
  public: void display(){ cout<<"\n variable defined in the base class, accessed in the derived class\t"<<a;}
};
void main()
{
        derivedd dobj;
        dobj.get();
        dobj.cal();
        dobj.display();
}
```

Multiple inheritance



Multiple Inheritance

- In this inheritance the derived class may inherit data from two or more base classes.
- All the members of the class declared either protected or public accessibility mode are accessible in the derived class.

Example

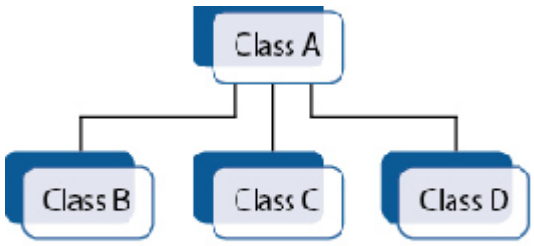
```
class stud
```

```
{ protected: int roll, m1, m2;
public: void get() { cout << "Enter the Roll No.: ";
cin >> roll;
 cout << "Enter the two highest marks: ";
cin >> m1 >> m2; }
 };
class extracurriculam
{ protected: int xm;
public:
 void getsm() { cout << "\nEnter the mark for Extra Curriculam Activities: ";
 cin >> xm; }
};
 class output : public stud, public extracurriculam
{ int tot, avg;
public: void display() { tot = (m1 + m2 + xm); avg = tot / 3;
 cout << "\n\n\tRoll No : " << roll << "\n\tTotal : " << tot; cout << "\n\tAverage : " <<
avg; }
 };
int main() { output O; O.get(); O.getsm(); O.display(); }
```

Hierarchical Inheritance

- multiple derived classes get inherited from a single base class.
- There will be one base class and more than one derived class in this type of inheritance.
- All the data members in the base class should be of the public access

mode.



Hierarchical Inheritance

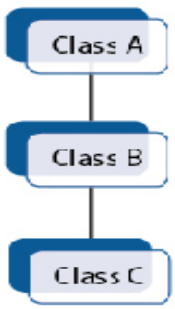
Example

```
class member
{ char gender[10]; int age;
public:
void get() { cout << "Age: "; cin >> age; cout << "Gender: "; cin >> gender; } void disp()
{ cout << "Age: " << age << endl; cout << "Gender: " << gender << endl; }
};
class stud : public member
{ char level[20];
public:
void getdata() { member::get(); cout << "Class: "; cin >> level; }
void disp2() { member::disp(); cout << "Level: " << level << endl; }
 };
class staff : public member
 { float salary;
public: void getdata() { member::get(); cout << "Salary: Rs."; cin >> salary; } void disp3()
{ member::disp(); cout << "Salary: Rs." << salary << endl; }
 };
```

```
int main() { member M; staff S; stud s;
 cout << "Student" << endl;
 cout << "Enter data" << endl; s.getdata();
cout << endl << "Displaying data" << endl;
s.disp();
cout << endl << "Staff Data" << endl; cout << "Enter data" << endl; S.getdata();
cout << endl << "Displaying data" << endl; S.disp();
}
```

Multilevel Inheritance

- In this type of inheritance derived class is again used as the base class for some other derived class.



Multilevel Inheritance

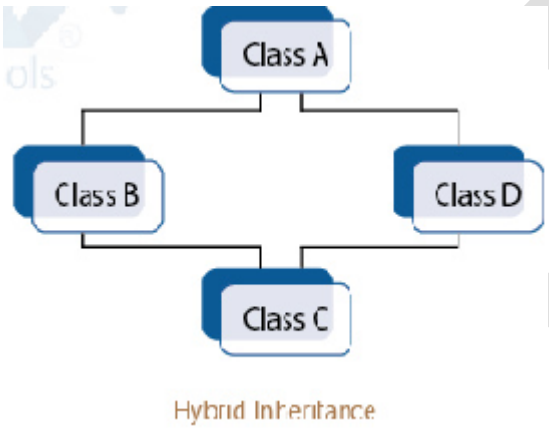
Example

```
class base
{ public:
 void display1() { cout << "\nBase class content."; }
 };
class derived : public base
{public:
```

```
void display2() { cout << "1st derived class content."; }
};
class derived2 : public derived
{  void display3() { cout << "\n2nd Derived class content."; }
};
int main() { derived2 D; //D.display3(); D.display2(); D.display1(); }
```

Hybrid inheritance

- Two or more types of inheritance are used in the code to bring about the hybrid inheritance.



Hybrid Inheritance

Example:

```
class arithmetic
{
protected:
int num1, num2;
public:
void getdata()
{
cout<<"For Addition:";
cout<<"\nEnter the first number: ";
```

```
cin>>num1;
cout<<"\nEnter the second number: ";
cin>>num2;
}
};
class plus:public arithmetic
{
protected:
int sum;
public:
void add()
{
sum=num1+num2;
}
};
class minus
{
protected:
int n1,n2,diff;
public:
void sub()
{
cout<<"\nFor Subtraction:";
cout<<"\nEnter the first number: ";
cin>>n1;
cout<<"\nEnter the second number: ";
cin>>n2;
diff=n1-n2;
}
};
class result:public plus, public minus
```

```
{
public:
void display()
{
cout<<"\nSum of "<<num1<<" and "<<num2<<"= "<<sum;
cout<<"\nDifference of "<<n1<<" and "<<n2<<"= "<<diff;
}
};
void main()
{
clrscr();
result z;
z.getdata();
z.add();
z.sub();
z.display();
getch();
}
```

C++

Unit IV

Inline functions

- The major aim of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times.
- Every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack and returning to the calling function.
- To eliminate the cost of calls to small functions, a new feature called inline function.
- An inline function is a function that is expanded in line when it is invoked. The compiler replaces the function call with the corresponding function code.

The inline functions are defined as follows

```
Inline function-header
{   function body;    }
```

Eg

```
inline double cube(double a)
{    return (a * a * a );    }
```

in some situations where inline function may not work

1. For functions returning values, if a loop, a switch or a goto exists.
2. For functions not returning values, if a return statement exists.
3. If function contain static variables.
4. If inline functions are recursive.

Note: Inline expansion makes a program run faster because the overhead of a function call and return is eliminated. It makes the program to take up more memory because the statements that define the inline function are reproduced at each point where the function is called.

```
Inline float mul(float x, float y)
{     return ( x * y );    }
inline double div(double p, double q)
{     return ( p / q );    }
int main()
{
flaot a = 12.345;
float b= 9.82;
cout<<mul(a,b);
```

```
cout<<div(a,b);
return 0;
}
```

Friend Functions

- A function declared as a friend is not in the scope of the class to which it has been declared as friend. It has full access to the private members of the class.
- To make an outside function "friendly" to a class, we have to simply declare this function a friend of the class as shown below

```
Class abc
{
……
……
Public:
……..
……..
friend void xyz(void);
};
```

- The function declaration should be preceded by the keyword friend.
- The function is defined elsewhere in the program like a normal C++ function.
- The function definition does not use either the keyword friend or the scope operator :: .
- The functions that are declared with the keyword friend are known as friend functions.
- A function can be declared as a friend in any number of classes. A friend function has full access right the private members of the class

A friend function possesses certain special characteristics

a. It is not in the scope of the class to which it has been declared as friend.
b. It cannot be called using the object of the class
c. It can be invoked like a normal function without the help of any object.
d. It cannot access the member names directly and has to use an object name and dot membership operator with each member name.
e. It can be declared either in the public or the private part of a class without affecting its meaning.
f. It has the objects as arguments.

```
Class sample
{
int a;
int b;
public:
void setvalue(){a=25;b=40}
friend float mean(sample s);
```

```
};
float mean(sample s)
{   return float(s.a + s.b) / 2.0;   }
int main()
{
sample x;
x.setvalue();
cout<<"Mean value       ="<<mean(x);
return 0;   }
```

Virtual function

- A virtual function a member function which is declared within base class and is re-defined (Overriden) by derived class.
- When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

1. They Must be declared in public section of class.
2. Virtual functions cannot be static and also cannot be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
4. The prototype of virtual functions should be same in base as well as derived class.
5. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
6. A class may have virtual destructor but it cannot have a virtual constructor.

Compile-time(early binding) VS run-time(late binding) behavior of Virtual Functions

Consider the following simple program showing run-time behavior of virtual functions.

```
// CPP program to illustrate
// concept of Virtual Functions
#include<iostream>
using namespace std;

class base
{
public:
   virtual void print ()
   { cout<< "print base class" <<endl; }
```

```
    void show ()
    { cout<< "show base class" <<endl; }
};

class derived:public base
{
public:
    void print ()
    { cout<< "print derived class" <<endl; }

    void show ()
    { cout<< "show derived class" <<endl; }
};

int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    //virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
}
```

Polymorphism

What is Polymorphism?

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.

Function overloading

- Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.
- Function overloading can be considered as an example of polymorphism feature in C++.

Following is a simple C++ example to demonstrate function overloading.

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

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
void print(char* c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

Operator Overloading

- For user-defined types (like: objects), you can redefine the way operator works.
- For example:

  If there are two objects of a class that contains string as its data members. You can redefine the meaning of + operator and use it to concatenate those strings.

- This technique is called operator overloading.

Example

```
        #include <iostream.h>
        Class Test
        {       private: int count;
                public:
                void operator ++()
                { count =count+1; }
                Void display()
                { cout<<"Count:  "<<count;}
        };
        int main()
{       Test t;
        ++t;
```

```
    t.display();
    return (0);
}
```

C++ Streams

- A stream is a sequence of bytes.
- It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent.
- The source stream that provides data to the program is called the input stream and the destination stream that receives output from the program is called the output stream.
- A program extracts the bytes from an input stream and inserts bytes into an output stream.
- The data in the input stream can come from the keyboard or any other storage device.
- The data in the output stream can go to the screen or any other storage device.
- A stream acts as an interface between the program and the input / output device.

Stream classes	Contents
Ios (General input/output stream class)	• contains basic facilities that are used by all other input and output classes. • Also contains a pointer to a buffer object. (streambuf object) • Declares constants and functions that are necessary for handling formatted input and output operations.
istream (input stream)	• Inherits the properties of ios • Declares input functions such as get(), getline() and read() • Contains overloaded extraction operator >>
Ostream(outpt stream)	• Inherits the properties of ios • Declares output functions such as put() and write() • Contains overloaded extraction operator <<
Iostream (input / output stream)	• Inherits the properties of ios istream and ostream through multiple inheritance and thus contains all the input and output fucnton.
Streambuf	• Provides an interface to physical devices through buffers. • Acts as a base for filebuf class used

	ios files.

Put() and get() functions

Get()

There are two types of get() functions.

1. get(char *) – assigns the input character to its argument.
2. get(void) used to fetch a character including blank space, tab and the new line character.

Syntax cin.get(c) – where c is a character

Put()

The put() function can be used to output a line of text character by character.

Syntax cout.put(ch) – ch is a character.

```
Int main()
{
int count = 0;
char c;
cout<<"Input text";
cin.get(c);
while(c!='\n')
{
cout.put(c)
count++;
cin.get(c);
}
cout<<"\n Number of characters  ="<<count;
return 0;
}
```

getline() and write() functions

The getline() function reads a whole line of text that ends with a new line character. This function can be invoked by using the object cin as follows.

```
cin.getline(line,size);
```

This function call invokes the function getline() which reads character input into the variable line.

```
Int main()
{
int size = 20;
```

```
char city[20];
cout<<"Enter city name:";
cin>>city;
cout<<"City name:"<<city;
cout<<"Enter city name again";
cin.getline(city,size);
cout<<"city name now:"<<city;
cout<<"Enter another city name:";
cin.getline(city,size);
cout<<"New city name:"<<city;
return 0;
}
```

write()

The write() function displays an entire line and has the following form

Cout.write(line,size);

The first argument line represents the name of the string to be displayed and the second argument size indicates the number of characters to display.

```
Int main()
{
char * string1 = "C++";
char * string2 = "Programming";
int m = strlen(string1);
int n=strlen(string2);
for(int i=1;i<n;i++)
{       cout.write(string2,i);   }
for(i=n;i>0;i--)
{       cout.write(string2,i);}
cout.write(string1,m).write(string2,n);
cout.write(string1,10);
return 0;
}
```

Formatted console I/O operations

C++ supports a number of features that could be used for formatting the output. These features include:

- ios class functions and flags.
- Manipulators
- User-defined output functions.

The ios class contains a large number of member functions that would help us to format the output in a number of ways.

ios format functions

Width()	To specify the required field size for displaying an output value

Precision()	To specify the number of digits to be displayed after the decimal point of a float value.
Fill()	To specify a character that is used to fill the unused portion of a field
Setf()	To specify format flags that can control the form of output display
Unsetf()	To clear the flags specified.

Manipulators:

Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream.

Manipulators	Equivalent ios function
Setw()	width()
Setprecision()	precision()
Setfill()	fill()
Setiosflags()	setf()
Resetiosflags()	unsetf()

Width()

- We can use the function to define the width of a field necessary for the output of an item.
- Cout.width(w);where w is the field width.
- The output will be printed in a field of w characters wide at the right end of the field.
- The width() function can specify the field width for only one item.
- After printing one item it will revert back to the default.

```
Cout.width(5);
```

Precision()

- The floating numbers are printed with six digits after the decimal point.
- We can specify the number of digits to be displayed after the decimal point while printing the floating point numbers.
- This can be done by using the precesion() member function as follows:
- Cout.precision(d); where d is the number of digits to the right of the decimal point.

```
Cout.precision(3);
Cout<<sqrt(2);
Cout<<3.141
```

Fill()

- The unused positions of the field are filled with white spaces, by default.
- We can use the fill() function to fill the unused positions by any desired character. It is used in the following form:
- Cout.fill(ch); where ch represents the character which is used for filling the unused positions.

Cout.fill('*');

Unit V

File Handling in C++

Working with files generally requires the following kinds of data communication methodologies:

- Data transfer between console units
- Data transfer between the program and the disk file

So far we have learned about iostream standard library which provides cin and cout methods for reading from standard input and writing to standard output respectively. In this chapter, you will get to know how files are handled using C++ program and what are the functions and syntax used to handle files in C++.

List of file handling classes

1. Ofstream: This file handling class in C++ signifies the output file stream and is applied to create files for writing information to files
2. Ifstream: This file handling class in C++ signifies the input file stream and is applied for reading information from files
3. Fstream: This file handling class in C++ signifies the file stream generally, and has the capabilities for representing both ofstream and ifstream

All the above three classes are derived from fstreambase and from the corresponding iostream class and they are designed specifically to manage disk files.

Opening and Closing a File in C++

If programmers want to use a disk file for storing data, they need to decide about the following things about the file and its intended use. These points that are to be noted are:

- A name for the file
- Data type and structure of the file
- Purpose (reading, writing data)
- Opening method
- Closing the file (after use)

Files can be opened in two ways. They are:

1. Using constructor function of the class

2. Using member function open of the class

Opening a File

The first operation generally performed on an object of one of these classes to use a file is the procedure known as to opening a file. An open file is represented within a program by a stream and any input or output task performed on this stream will be applied to the physical file associated with it. The syntax of opening a file in C++ is:
open (filename, mode);

There are some mode flags used for file opening. These are:

- ios::app: append mode
- ios::ate: open a file in this mode for output and read/write controlling to the end of the file
- ios::in: open file in this mode for reading
- ios::out: open file in this mode for writing
- ios::trunk: when any file already exists, its contents will be truncated before file opening

Closing a file in C++

When any C++ program terminates, it automatically flushes out all the streams releases all the allocated memory and closes all the opened files. But it is good to use the close() function to close the file related streams and it is a member of ifsream, ofstream and fstream objects.
The structure of using this function is:
void close();

General functions used for File handling

1. open(): To create a file
2. close(): To close an existing file
3. get(): to read a single character from the file

4. put(): to write a single character in the file
5. read(): to read data from a file
6. write(): to write data into a file

Reading from and writing to a File

While doing C++ program, programmers write information to a file from the program using the stream insertion operator (<<) and reads information using the stream extraction operator (>>). The only difference is that for files programmers need to use an ofstream or fstream object instead of the cout object and ifstream or fstream object instead of the cin object.

Example:

```
#include <iostream>

#include <fstream.h>

void main () {

  ofstream file;

  file.open ("egone.txt");

  file << "Writing to a file in C++....";

  file.close();

  getch();

}
```

Another Program for File Handling in C++

Example:

```
#include <iostream>

#include<fstream.h>

void main()

{

  char c, fn[10];
```

```
  cout<<"Enter the file name....:";
  cin>>fn;
  ifstream in(fn);
  if(!in)
  {
    cout<<"Error! File Does not Exist";
    getch();
    return;
  }
  cout<<endl<<endl;
  while(in.eof()==0)
  {
    in.get(c);
    cout<<c;
  }
  getch();
}
#include<fstream.h>
#include<iostream.h>

int main()
{
 char data[100];
  ofstream  outfile;  //object creation for ofstream class
  outfile.open("lg.txt",ios::app|ios::out); // opening existing file in append
mode/update mode
```

```
  cout<<"\n Writing to a file "<< endl;
cout<<"\n Enter your name : ";
cin.getline(data,100);
outfile<<data<<endl;     // inserting data into the new file
cout<<"\nEnter your age : ";
cin>>data;
outfile<<data<<endl;     // inserting data into the new file
outfile.close();         // close new file pointer.
return(0);
}
#include<fstream.h>
#include<conio.h>
#include<iostream.h>
#include<cstdio.h>
```

Write from file

```
int main()
{
file *fp;
  char data[100];
   clrscr();
   ifstream infile;

 fp= infile.open("lg.txt");
  cout<<"reading from the fill lg.txt"<<endl;
  while(!feof(fp))
  {
  infile>>data;
 cout<<data<<endl;
infile>>data;
cout.width(5);
```

```
cout.fill('*');
cout<<data<<endl;
}
infile.close();
getch();
return(0);
}
```

Error handling during file operations

It is quite common that errors may occur while reading data from a file in C++ or writing data to a file. For example, an error may arise due to the following:

- When trying to read a file beyond indicator.
- When trying to read a file that does not exist.
- When trying to use a file that has not been opened.
- When trying to use a file in an appropriate mode i.e., writing data to a file that has been opened for reading.
- When writing to a file that is write-protected i.e., trying to write to a read-only file.

Failure to check for errors then the program may behave abnormally therefore an unchecked error may result in premature termination for the program or incorrect output.

Below are some Error handling functions during file operations in C/C++:

ferror():

The **ferror()** function checks for any error in the stream. It returns a value zero if no error has occurred and a non-zero value if there is an error.

```
#include<fstream.h>
#include<conio.h>
#include<iostream.h>
#include<stdio.h>
#include<stdlib.h>


void main()
{
 FILE*  fp;
 fp=fopen("nofile23.txt","w");       //error handling during file operation
 if(ferror(fp)==0)
 {
  cout<<"\nError!! File does not exist";
```

```
  }
  else
  {
    cout<<"\nfile pointer successfully fetched in memory";
  }
  fclose(fp);
 getch();
}
```

Error handling functions

Function	Return value and meaning
eof()	Returns true if end-of-file is encountered while reading; otherwise returns false
Fail()	Returns true when an input or output operation has failed
Bad()	Returns true if an invalid operations is attempted or any unrecoverable error has occurred. However if it is false, it may be possible to recover from any other error reported and continue operation.
Good()	Returns true if no error has occurred. This means, all the above functions are false. For instance, if file.good() is true, all is well with the stream file and we can proceed to perform I/O operations. When it returns false, no further operations can be carried out.

Command Line Arguments

he most important function of C/C++ is main() function. It is mostly defined with a return type of int and without parameters :

```
int main() { /* ... */ }
```

We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems.
To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

int main(int argc, char *argv[]) { /* ... */ }

or

int main(int argc, char **argv) { /* ... */ }

- **argc (ARGument Count)** is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
- The value of argc should be non negative.
- **argv(ARGument Vector)** is array of character pointers listing all the arguments.
- If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
- Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

```
#include <iostream>

using namespace std;

int main(int argc, char** argv)

cout << "You have entered " << argc

      << " arguments:" << "\n";

  for (int i = 0; i < argc; ++i)

      cout << argv[i] << "\n";

           return 0;}}
```

Input:

$ g++ mainreturn.cpp -o main

$ ./main geeks for geeks

utput:

You have entered 4 arguments:

./main

Properties of Command Line Arguments:

1. They are passed to main() function.

2. They are parameters/arguments supplied to the program when it is invoked.
3. They are used to control program from outside instead of hard coding those values inside the code.
4. argv[argc] is a NULL pointer.
5. argv[0] holds the name of the program.
6. argv[1] points to the first command line argument and argv[n] points last argument.

Note : You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes “” or single quotes ”.