THE ORIGIN OF C++

In 1979, a young engineer at Bell (now AT&T) Labs, Bjarne Stroustrup, started to experiment with extensions to C to make it a better tool for implementing large-scale projects. In those days, an average project consisted of tens of thousands of lines of code (LOC).

In 1983, several modifications and extensions had already been made to C with classes. In that year, the name "C++" was coined. Ever since then, the ++ suffix has become a synonym for object-orientation. (Bjarne Stroustrup could have made a fortune only by registering ++ as a trademark) It was also in that year that C++ was first used outside AT&T Labs. The number of users was doubling every few months -- and so was the number of compilers and extensions to the language.

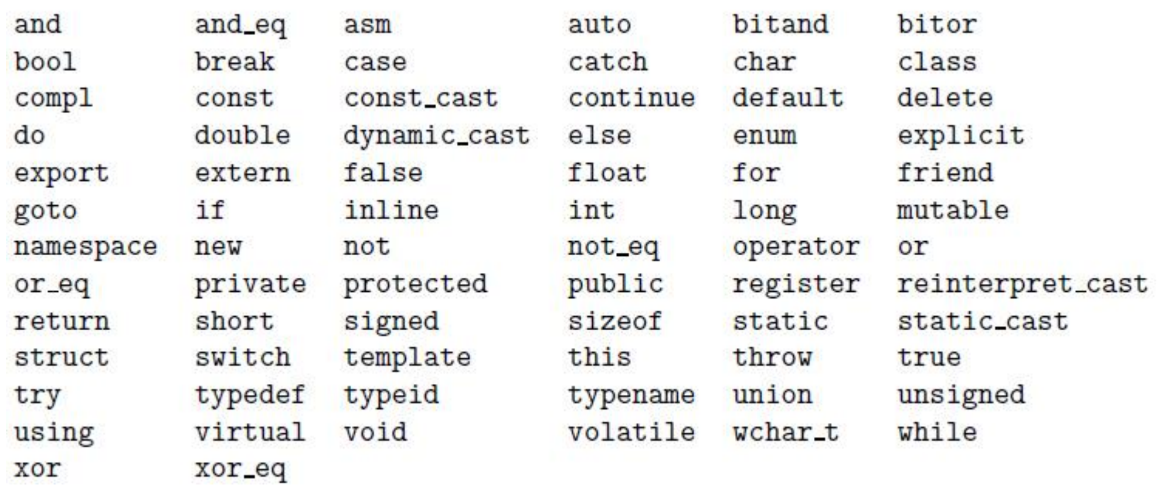
Between 1985 and 1989, C++ underwent a major reform. Protected members, protected inheritance, templates, and a somewhat controversial feature called multiple inheritance were added to the language. It was clear that C++ needed to become standardized.

BASIC INPUT/OUTPUT

CIN>>: It is used for input like scanf( ) in C.”>>” This is called insertors.

COUT<<: It is used for output like printf( ) in C.”<<” This is called exertors.

KEYWORDS



and	and_eq	asm	auto	bitand	bitor
bool	break	case	catch	char	class
compl	const	const_cast	continue	default	delete
do	double	dynamic_cast	else	enum	explicit
export	extern	false	float	for	friend
goto	if	inline	int	long	mutable
namespace	new	not	not_eq	operator	or
or_eq	private	protected	public	register	reinterpret_cast
return	short	signed	sizeof	static	static_cast
struct	switch	template	this	throw	true
try	typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	wchar_t	while
xor	xor_eq				

USAGE OF C++

By the help of C++ programming language, we can develop different types of secured and robust applications:

- Window application
- Client-Server application
- Device drivers
- Embedded firmware etc

Example:

```
#include<iostream.h>
void main( )
{
cout<<”Welcome”;
}
```

Example:

```
#include<iostream.h>
void main( )
{
cout<<”Welcome”<<endl;
cout<<”Hello”;
}
```

Example:

```
#include<iostream.h>
#include<conio.h>
void main( )
{
int num;
cout<<”Enter Number”
cin>>num;
cout<<num
getch( );
}
```

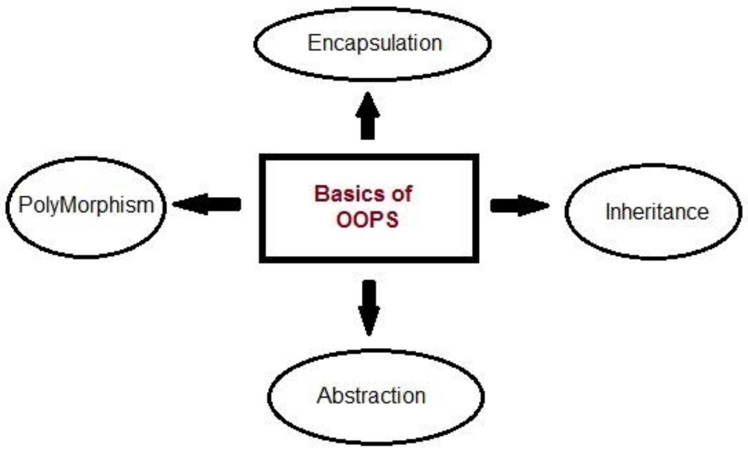
Example:

```
#include<iostream.h>
#include<conio.h>
void main( )
{
int num1,num2;
cout<<”Enter Two Numbers”;
cin>>num1>>num2;
cout<<num1+num2;
getch( );
```

}

OBJECT ORIENTED PROGRAMMING

C++ can be said to be as C language with classes. In C++ everything revolves around object of class, which have their methods & data members.



For Example : We consider human body as a class, we do have multiple objects of this class, with variable as color, hair etc. and methods as walking, speaking etc.

Main features of object oriented programming are-

1. Objects
2. Classes
3. Abstraction
4. Encapsulation
5. Inheritance
6. Overloading
7. Exception Handling

OBJECTS

Objects are the basic unit of OOP. They are instances of class, which have data members and uses various member functions to perform tasks.

CLASS

It is similar to structures in C language. Class can also be defined as user defined data type but it also contains functions in it. So, class is basically a blueprint for object. It declare & defines what data variables the object will have and what operations can be performed on the class's object.

ABSTRACTION

Abstraction refers to showing only the essential features of the application and hiding the details. In C++, classes provide methods to the outside world to access & use the data variables, but the variables are hidden from direct access.

ENCAPSULATION

It can also be said data binding. Encapsulation is all about binding the data variables and functions together in class.

INHERITANCE

Inheritance is a way to reuse once written code again and again. The class which is inherited is called base calls & the class which inherits is called derived class. So when, a derived class inherits a base class, the derived class can use all the functions which are defined in base class, hence making code reusable.

POLYMORPHISM

Polymorphion makes the code more readable. It is a features, which lets is create functions with same name but different arguments, which will perform differently. That is function with same name, functioning in different

OVERLOADING

Overloading is a part of polymorphion. Where a function or operator is made & defined many times, to perform different functions they are said to be overloaded.

EXCEPTION HANDLING

Exception handling is a feature of OOP, to handle unresolved exceptions or errors produced at runtime.

DATA TYPES

There are following data types:

Type	**Keyword**
Boolean	bool
Character	char
Integer	int
Floating point	float
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

SIZEOF( )- It tells about how much space is taken by a variable.

```
#include <iostream.h>
int main()
{   cout << "Size of char : " << sizeof(char) << endl;
   cout << "Size of int : " << sizeof(int) << endl;
   cout << "Size of short 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;
}
```

TYPEDEF DECLARATIONS:

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, the following tells the compiler that feet is another name for int:

```
typedef int feet;
```

example:

```
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. To create 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;
```

Here, 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, 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.

COMMENTS

The C++ comments are statements that are not executed by the compiler. The comments in C++ programming can be used to provide explanation of the code, variable, method or class. By the help of comments, you can hide the program code also.

There are two types of comments in C++.

- Single Line comment
- Multi Line comment

Single Line Comment

The single line comment starts with // (double slash).

```
#include <iostream.h>
int main()
{
 int x = 11; // x is a variable
 cout<<x<<"\n";
}
```

Multi Line Comment

The C++ multi line comment is used to comment multiple lines of code. It is surrounded by slash and asterisk (/∗ ..... ∗ /).

```
#include <ostream.h>
int main()
{
/* declare and
print variable in C++. */
 int x = 35;
 cout<<x<<"\n";
}
```

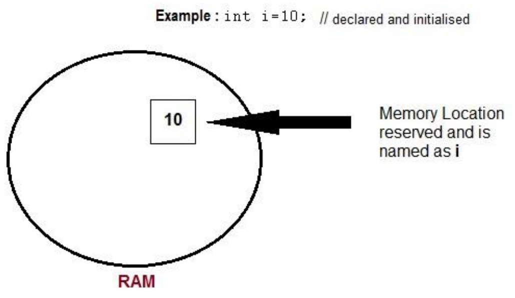
STANDARD END LINE (endl)

The **endl** is a predefined object of **ostream** class. It is used to insert a new line characters and flushes the stream.

```
#include <iostream.h>
int main( ) {
cout << "Welcome"<<endl;
cout << "Here";
cout << "End of line"<<endl;
}
```

VARIABLES

Variable are used in C++, where we need storage for any value, which will change in program. Variable can be declared in multiple ways each with different memory requirements and functioning. Variable is the name of memory location allocated by the compiler depending upon the datatype of the variable.


Example : int i=10; // declared and initialised
10
Memory Location reserved and is named as i
RAM

BASIC TYPES OF VARIABLES

Each variable while declaration must be given a datatype, on which the memory assigned to the variable depends. Following are the basic types of variables,

`bool`	For variable to store boolean values(True or False)
`char`	For variables to store character types.
`int`	for variable with integral values
`float` and `double` are also types for variables with large and floating point values	

DECLARATION AND INITIALIZATION

Variable must be declared before they are used. Usually it is preferred to declare them at the starting of the program, but in C++ they can be declared in the middle of program too, but must be done before using them.

Example :

```
int i;      // declared but not initialized
char c;
int i, j, k;  // Multiple declaration
```

Initialization means assigning value to an already declared variable,

```
int i;   // declaration
i = 10;  // initialization
```

Initialization and declaration can be done in one single step also,

```
int i=10;         //initialization and declaration in same step
```

int i=10, j=11;
If a variable is declared and not initialized by default it will hold a garbage value. Also, if a variable is once declared and if try to declare it again, we will get a compile time error.

SCOPE OF VARIABLES

All the variables have their area of functioning, and out of that boundary they don't hold their value, this boundary is called scope of the variable. For most of the cases its between the curly braces,in which variable is declared that a variable exists, not outside it. We will study the storage classes later, but as of now, we can broadly divide variables into two main types,

1.Global Variables
2. Local variables

GLOBAL VARIABLES

Global variables are those, which ar once declared and can be used throughout the lifetime of the program by any class or any function. They must be declared outside the main() function. If only declared, they can be assigned different values at different time in program lifetime. But even if they are declared and initialized at the same time outside the main() function, then also they can be assigned any value at any point in the program.

Example : Only declared, not initialized

```
include <iostream>
int x;              // Global variable declared
int main()
{ x=10;               // Initialized once
 cout <<"first value of x = "<< x;
 x=20;               // Initialized again
 cout <<"Initialized again with value = "<< x;
}
```

LOCAL VARIABLES

Local variables are the variables which exist only between the curly braces, in which its declared. Outside that they are unavailable and leads to compile time error.
Example

```
int main( )
{ int i=10;
 if(i<20)       // if condition scope starts
  {    int n=100;   // Local variable declared and initialized
  }            // if condition scope ends
 cout << n;      // Compile time error, n not available here
}
```

OPERATORS

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provides the following types of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators
-

ARITHMETIC OPERATORS

There are following arithmetic operators supported by C++ language:

Assume variable A holds 10 and variable B holds 20, then:

Operator	**Description**	**Example**
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

RELATIONAL OPERATORS

There are following relational operators supported by C++ language
Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

LOGICAL OPERATORS

There are following logical operators supported by C++ language. Assume variable A holds 1 and variable B holds 0, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

BITWISE OPERATORS

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows:

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format they will be as follows:

```
A = 0011 1100
B = 0000 1101
-----------------
A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A  = 1100 0011
```

The Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

<u>ASSIGNMENT OPERATORS</u>

There are following assignment operators supported by C++ language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2

&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

MISC OPERATORS

There are few other operators supported by C++ Language.

Operator	**Description**
sizeof	sizeof operator returns the size of a variable. For example, sizeof(a), where a is integer, will return 4.
Condition ? X : Y	Conditional operator. If Condition is true ? then it returns value X : otherwise value Y
,	Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
. (dot) and -> (arrow)	Member operators are used to reference individual members of classes, structures, and unions.
Cast	Casting operators convert one data type to another. For example, int(2.2000) would return 2.
&	Pointer operator & returns the address of an variable. For example &a; will give actual address of the variable.
*	Pointer operator * is pointer to a variable. For example *var; will pointer to a variable var.

LOOP

A loop statement allows us to execute a statement or group of statements multiple times.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body
nested loops	You can use one or more loop inside any another while, for or do..while loop.

WHILE LOOP

while loop can be address as an entry control loop. It is completed in 3 steps.

1. Variable initialization.(e.g int x=0;)
2. condition(e.g while(x<=10))
3. Variable increment or decrement (x++ or x-- or x=x+2)

Syntax :

```
variable initialization ;
while (condition)
{
 statements ;
 variable increment or decrement ;
}
```

FOR LOOP

for loop is used to execute a set of statement repeatedly until a particular condition is satisfied. we can say it an open ended loop. General format is,

```
for(initialization; condition ; increment/decrement)
{
  statement-block;
}
```

In for loop we have exactly two semicolons, one after initialization and second after condition. In this loop we can have more than one initialization or increment/decrement, separated using comma operator. for loop can have only one condition.

DO WHILE LOOP

In some situations it is necessary to execute body of the loop before testing the condition. Such situations can be handled with the help of do-while loop. do statement evaluates the body of the loop first and at the end, the condition is checked using while statement. General format of do-while loop is,

```
do
{
 ....
 .....
}
while(condition)
```

NESTED FOR LOOP

We can also have nested for loop, i.e one for loop inside another for loop. Basic syntax is,

```
for(initialization; condition; increment/decrement)
{
  for(initialization; condition; increment/decrement)
     {
        statement ;
     }
}
```

JUMPING OUT OF LOOP

Sometimes, while executing a loop, it becomes necessary to skip a part of the loop or to leave the loop as soon as certain condition becocmes true, that is jump out of loop. C language allows jumping from one statement to another within a loop as well as jumping out of the loop.

1) break statement

When break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.

2) <u>continue statement</u>

It causes the control to go directly to the test-condition and then continue the loop process. On encountering continue, cursor leave the current cycle of loop, and starts with the next cycle.

<u>BRANCHING/DECISION STATEMENTS</u>

Statement	**Description**
if statement	An if statement consists of a boolean expression followed by one or more statements.
if...else statement	An if statement can be followed by an optional else statement, which executes when the boolean expression is false.
switch statement	A switch statement allows a variable to be tested for equality against a list of values.
nested if statements	You can use one if or else if statement inside another if or else if statement(s).
nested switch statements	You can use one swicth statement inside another switch statement(s).

CLASS

A Class is way to bind the data and its associated functions together. It allows the data and functions to be hidden, if necessary, from external use. When defining a class, we are creating a new abstract data type that can be treated like any other built-in data type. The class declaration describes the type and scope of its members. The general form of declaration is as follows:

Syntax:

```
class class_name
{
Private:
variable declarations;
function declarations;
public:
variable declaration;
function declaration;
};
```

The class declaration is similar to struct declaration. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively known as class members.

OBJECT

Objects are the basic run time entity in an object-oriented system. They may represent a person, a place, a bank account or any that the program has to handle. They may also represent user defined data such as vectors, time, and lists. When a program is executed, the object interacted by sending messages to one another. **Example:**

```
class emp
{char name;
float age;
public:
void getdata( )
{cout<<”ENTER NAME:”;
cin>>name;
cout<<”ENTER AGE:”;
cin>>age;
}
void putdata( )
{cout<<”NAME:”<<name;
cout<<”AGE:”<<age;
}
};
int main( )
```

```
{emp e;                    // Here e is object
e.getdata( );
e.putdata( );
return 0;
}
```

ACCESS MODIFIERS

Access modifiers are used to implement an important feature of Object Oriented Programming known as **Data Hiding**.

Access modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.
There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be **Private**.

Public: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

```
#include<iostream.h>
class Circle
{   public:
       double radius;
       double  compute_area()
       {
          return 3.14*radius*radius;
       }
};

int main()
{  Circle obj;
   obj.radius = 5.5;         // accessing public datamember outside class
   cout << "Radius is:" << obj.radius << "\n";
   cout << "Area is:" << obj.compute_area();
   return 0;
}
```

Private: The class members declared as **private** can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.

Example:
```
#include<iostream.h>
class Circle
{
     private:
        double radius;
    public:
        double  compute_area()
        {
           return 3.14*radius*radius;
        }

};

int main()
{
    Circle obj;
    obj.radius = 1.5;       // Error
    cout << "Area is:" << obj.compute_area();
    return 0;
}
```

Protected: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

```
#include <iostream.h>
class Parent
{   protected:
    int id_protected;

};

class Child : public Parent
{  public:
    void setId(int id)
    {   id_protected = id;
    }
    void displayId()
    {   cout << "id_protected is:" << id_protected << endl;
    }
};

int main( )
{
    Child obj1;
    obj1.setId(81);
    obj1.displayId();
    return 0;}
```

CONSTRUCTORS

A constructor is used to initialize an object. A default constructor is one that can be invoked without any arguments. If there is no user-declared constructor for a class, and if the class does not contain `const` or reference data members, the implementation implicitly declares a default constructor for it.

DEFAULT CONSTRUCTOR

This constructor has no arguments in it. Default Constructor is also called as *no argument constructor*.

For example:

```
class Exforsys
{   private:
        int a,b;
    public:
        Exforsys( );
        {cout<<"welcome"
        }
};

void main()
{
Exforsys ex;

}
```

PARAMETERIZED CONSTRUCTOR

This constructor has some arguments in it. It is called *parameterized constructor*.

```
#include<iostream.h>
class Tomato
{int m,n;
public:
Tomato(int x, int y)                //constructor defined
{
  m=x;
  n=y;
}
void display( )
{
cout<<"m="<<m<<"\n";
cout<<"n="<<n<<"\n";
}
};
```

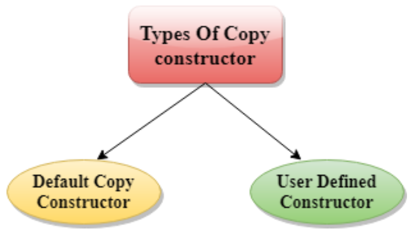
```
int main( )
{
Tomato t1(10,100);
Tomato t2 (25,75);
cout<<"\n OBJECT1"<<"\n";
t1.display( );
cout<<"\n OBJECT2"<<"\n";
t2.display( );
return 0;
}
```

COPY CONSTRUCTOR

A Copy constructor is an **overloaded** constructor used to declare and initialize an object from another object.

Copy Constructor is of two types:

- **Default Copy constructor:** The compiler defines the default copy constructor. If the user defines no copy constructor, compiler supplies its constructor.
- **User Defined constructor:** The programmer defines the user-defined constructor.



User-defined Copy Constructor

```
#include <iostream.h>
class A
{
  public:
   int x;
   A(int a)              // parameterized constructor.
   { x=a;
   }
   A(A &i)               // copy constructor
   {  x = i.x;
   }
};
int main()
{
  A a1(20);              // Calling the parameterized constructor.
 A a2(a1);               //  Calling the copy constructor.
 cout<<a2.x;
  return 0;
}
```

When Copy Constructor is called

Copy Constructor is called in the following scenarios:

- When we initialize the object with another existing object of the same class type. For example, Student s1 = s2, where Student is the class.
- When the object of the same class type is passed by value as an argument.
- When the function returns the object of the same class type by value.

DESTRUCTOR

Destructors are also special member functions used in C++ programming language. Destructors have the opposite function of a constructor. The main use of destructors is to release dynamic allocated memory. Destructors are used to free memory, release resources and to perform other clean up. Destructors are automatically named when an object is destroyed. Like constructors, destructors also take the same name as that of the class name.

General Syntax of Destructors

~ classname();

```
#include<iostream.h>
class Mango
{   public:
      Mango ( )                    //constructors
        {cout<<"Constructing Object";
        }

     ~Mango( )                  //destructor
        {
       cout<<"Destructing Object";
       }
};

void main( )
{  Mango obj( );
}
```

CONSTRUCTORS AND DESTRUCTORS SHOULD BE MINIMAL

When you are designing a class, remember that it might serve as a base for other subclasses. It can also be used as a member object in a larger class. As opposed to ordinary member functions, which can be overridden or simply not called, the base class constructor and destructor are automatically invoked. It is not a good idea to force users of a derived and embedded object to pay for what they do not need, but are forced to accept. In other words, constructors and destructors should contain nothing but the minimal functionality needed to construct an object and destroy it.

ENCAPSULATION

The packaging of data values and member functions within one object is called an encapsulation. It is also called data hiding which helps to maintain the integrity of the object. It saves the data from misuse and outside interference. The data cannot be accessed directly but access controls can be specified in order to obtain the information. The data or object can be made public or private depending on the needs. The data which is private is not accessible outside the scope of the object. When the data is public it can be accessed by the other parts of the program.

Example:

```
#include<iostream>
class rectangle
{
  private:
                double length;
        public:
                double breadth;
                double area( )
                {
                        return(length*breadth);
                }
                double len( )
                {
                        return(length);
                }
                rectangle(double lenght1,double breadth1)
                {
                        length=lenght1;
                        breadth=breadth1;
                }
};

 int main( )
{
        rectangle r1(3.5,4.6);
        double a=r1.len( );
        double b=r1.breadth;
        cout << "The lenght is : " << a <<  endl;
        cout << "The breadth is : " << b << endl;
        cout << "The area is : " << r1.area( ) << endl;
        return(0);
}
```

INHERITANCE

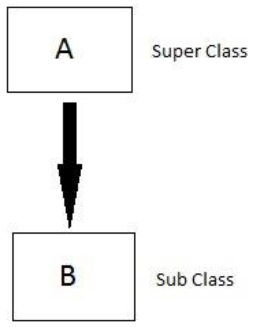
When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class and this phenomenon is called “Inheritance”.

There are five types of inheritances. They are following-

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchal Inheritance
5. Hybrid Inheritance

SINGLE INHERITANCE

When a class can inherit members from only one class is called “SINGLE INHERITANCE”. There are two classes one becomes base class which is called and the caller class is called drive class.



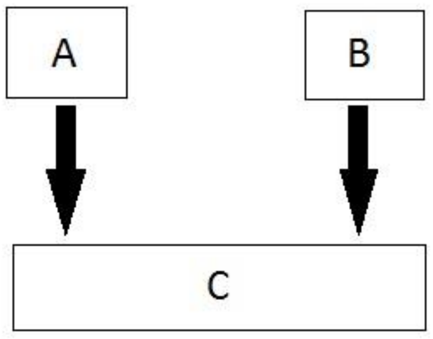
Example:

```
#include <iostream.h>
class Shape
{   public:
      void setWidth(int w)
      {  width = w;
      }
      void setHeight(int h)
      {  height = h;
      }
   protected:
      int width;
```

```
    int height;
};

class Rectangle: public Shape
{   public:
    int getArea()
    {   return (width * height);
    }
};
int main(void)
{ Rectangle Rect;
  Rect.setWidth(5);
  Rect.setHeight(7);
  cout << "Total area: " << Rect.getArea() << endl;
  return 0;
}
```

MULTIPLE INHERITANCE

When a class can inherit members from more than one class is called "MULTIPLE INHERITANCE".



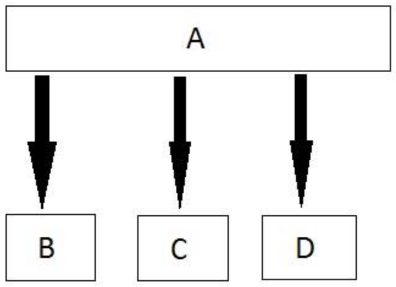
Example:

```
#include <iostream.h>
// Base class Shape
class Shape
{   public:
    void setWidth(int w)
    {       width = w;
    }
```

```
      void setHeight(int h)
      {         height = h;
      }
   protected:
      int width;
      int height;
};
// Base class PaintCost
class PaintCost
{   public:
      int getCost(int area)
      {
         return area * 70;
      }
};
// Derived class
class Rectangle: public Shape, public PaintCost
{   public:
      int getArea()
      {         return (width * height);
      }
};

int main(void)
{ Rectangle Rect;
   int area;
  Rect.setWidth(5);
   Rect.setHeight(7);
   area = Rect.getArea();
   cout << "Total area: " << Rect.getArea() << endl;
   cout << "Total paint cost: $" << Rect.getCost(area) << endl;
   return 0;
}
```

HIERARCHICAL INHERITANCE

In this type of inheritance, multiple derived classes inherits from a single base class.

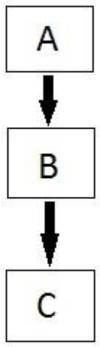


```
#include <iostream.h>
#include <conio.h>

class person
{
   char name[100],gender[10];
   int age;
   public:
      void getdata( )
      {
         cout<<"Name: ";
         fflush(stdin);                                    /*clears input stream*/
         gets(name);
         cout<<"Age: ";
         cin>>age;
         cout<<"Gender: ";
         cin>>gender;
      }
      void display( )
      {
         cout<<"Name: "<<name<<endl;
         cout<<"Age: "<<age<<endl;
         cout<<"Gender: "<<gender<<endl;
      }
};

class student: public person
{
   char institute[100], level[20];
   public:
      void getdata( )
      {
         person::getdata( );
         cout<<"Name of College/School: ";
         fflush(stdin);
         gets(institute);
         cout<<"Level: ";
         cin>>level;
      }
      void display( )
      {
         person::display( );
         cout<<"Name of College/School: "<<institute<<endl;
         cout<<"Level: "<<level<<endl;
      }
};
```

```
class employee: public person
{
   char company[100];
   float salary;
   public:
      void getdata( )
      {
         person::getdata( );
         cout<<"Name of Company: ";
         fflush(stdin);
         gets(company);
         cout<<"Salary: Rs.";
         cin>>salary;
      }
      void display( )
      {
         person::display( );
         cout<<"Name of Company: "<<company<<endl;
         cout<<"Salary: Rs."<<salary<<endl;
      }
};

int main( )
{
   student s;
   employee e;
   cout<<"Student"<<endl;
   cout<<"Enter data"<<endl;
   s.getdata( );
   cout<<endl<<"Displaying data"<<endl;
   s.display( );
   cout<<endl<<"Employee"<<endl;
   cout<<"Enter data"<<endl;
   e.getdata( );
   cout<<endl<<"Displaying data"<<endl;
   e.display( );
   getch( );
   return 0;
}
```

MULTILEVEL INHERITANCE

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.

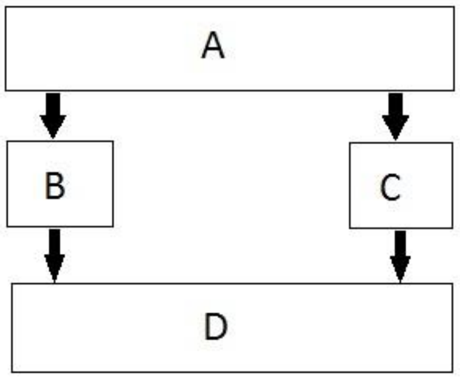


```
#include <iostream.h>
#include <conio.h>
class person
{
   char name[100],gender[10];
   int age;
   public:
      void getdata( )
     {
         cout<<"Name: ";
         fflush(stdin);                              /*clears input stream*/
         gets(name);
         cout<<"Age: ";
         cin>>age;
         cout<<"Gender: ";
         cin>>gender;
      }
      void display( )
      {
         cout<<"Name: "<<name<<endl;
         cout<<"Age: "<<age<<endl;
         cout<<"Gender: "<<gender<<endl;
      }
};
```

```
class employee: public person
{
    char company[100];
    float salary;
    public:
        void getdata()
        {
            person::getdata();
            cout<<"Name of Company: ";
            fflush(stdin);
            gets(company);
            cout<<"Salary: Rs.";
            cin>>salary;
        }
        void display()
        {
            person::display();
            cout<<"Name of Company: "<<company<<endl;
            cout<<"Salary: Rs."<<salary<<endl;
        }
};

class programmer: public employee
{
    int number;
    public:
        void getdata()
        {
            employee::getdata();
            cout<<"Number of programming language known: ";
            cin>>number;
        }
        void display()
        {
            employee::display();
            cout<<"Number of programming language known: "<<number;
        }
};

int main()
{
    programmer p;
    cout<<"Enter data"<<endl;
    p.getdata();
    cout<<endl<<"Displaying data"<<endl;
    p.display();
    getch();
    return 0;
}
```

HYBRID (VIRTUAL) INHERITANCE

Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



```
#include <iostream.h>
class A
{         public:
          int x;
};
class B : public A
{         public:
          B( )                              //constructor to initialize x in base class A
          {
            x = 10;
          }
};
class C
 {        public:
          int y;
          C( )                                  //constructor to initialize y
          {
             y = 4;
       }
};
class D : public B, public C               //D is derived from class B and class C
{
          public:
          void sum( )
          {
             cout << "Sum= " << x + y;
          }
};

int main()
{         D obj1;                               //object of derived class D
          obj1.sum();
          return 0;
}
```

PLOYMORPHISM

Polymorphism means having multiple forms of one thing. In inheritance, polymorphism is done, by method overriding, when both super and sub class have member function with same declaration bu different definition. 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.

Example:

```
#include <iostream.h>
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 << "Triangle class area :" <<endl;
      return (width * height / 2);
    }
};
// Main function for the program
int main( )
{
  Shape *shape;
  Rectangle rec(10,7);
  Triangle  tri(10,5);
  shape = &rec;
  shape->area();
  shape = &tri;
  shape->area();
  return 0;
}
```

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.

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.

Example:

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

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

PURE VIRTUAL FUNCTION

Pure Virtual Function is a Virtual function with no body. 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.

A virtual function is not used for performing any task. It only serves as a placeholder. When the function has no definition, such function is known as "do-nothing" function.The "do-nothing" function is known as a pure virtual function. A pure virtual function is a function declared in the base class that has no definition relative to the base class. A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes. The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

General Syntax of Pure Virtual Function

```
class classname //This denotes the base class of C++ virtual function
{
public:
virtual void virtualfunctioname() = 0 // pure virtual function
};
```

Example:

```
#include <iostream.h>
class Base {
public:
      virtual void f(int) = 0;
};
class Class1 : public Base {
public:
      virtual void f(int) {cout << "Class1" << endl;}
};
class Class2 : public Base {
public:
      virtual void f(int) {cout << "Class2" << endl;}
};
void  main( )
{    Base* p;
     Class1 c1;              Class2 c2;
      p = &c1;
     p->f(37);
      p = &c2;
      p->f(47);
}
```

Class1 and Class2 have no derivation or inheritance relationship with each other. They are independent classes with a common base.

FRIEND FUNCTIONS

At times you will want to grant this level of access not to an entire class, but only to one or two functions of that class. You can do this by declaring the member functions of the other class to be friends, rather than declaring the entire class to be a friend. In fact, you can declare any function, whether or not it is a member function of another class, to be a friend function.

Declare a function to be a friend by using the keyword `friend` and then the full specification of the function. Declaring a function to be a friend does not give the friend function access to your `this` pointer, but it does provide full access to all private and protected member data and functions.

Example:
```
#include <iostream.h>
class Box
{
  double width;
public:
  void show( )
    {
     cout<<"The Width is:"<<width;
    }
  friend void getPrivate( );
  };
}
// Note: getPrivate( ) is not a member function of any class.
void getPrivate ( )
{
   b1.width=89.989897;
}

int main( )
{
  Box b1;
  getPrivate( );
  b1.show( )
  return 0;
}
```

Following are some important points about friend functions and classes:
1) Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.
2) Friendship is not mutual. If a class A is friend of B, then B doesn’t become friend of A automatically.
3) Friendship is not inherited
4) The concept of friends is not there in Java.

INLINE FUNCTION

Inline functions are functions where the call is made to inline functions. The actual code then gets placed in the calling program.

Need of Inline Function:

Normally, a function call transfers the control from the calling program to the function and after the execution of the program returns the control back to the calling program after the function call. These concepts of function saved program space and memory space are used because the function is stored only in one place and is only executed when it is called. This concept of function execution may be time consuming since the registers and other processes must be saved before the function gets called.

The extra time needed and the process of saving is valid for larger functions. If the function is short, the programmer may wish to place the code of the function in the calling program in order for it to be executed. This type of function is best handled by the inline function. In this situation, the programmer may be wondering “why not write the short code repeatedly inside the program wherever needed instead of going for inline function?” Although this could accomplish the task, the problem lies in the loss of clarity of the program. If the programmer repeats the same code many times, there will be a loss of clarity in the program. The alternative approach is to allow inline functions to achieve the same purpose, with the concept of functions.

Example:

The concept of inline functions:

```
#include <iostream.h>
int exforsys(int);
void main( )
{
int x;
cout << “\n Enter the Input Value: ”;
cin>>x;
cout<<”\n The Output is: “ << exforsys(x);
}

inline int exforsys(int x1)
{
return 5*x1;
}
```

The output of the above program is:

```
Enter the Input Value: 10
The Output is: 50
```

ABSTRACT CLASS

A Class containing abstract methods is called an abstract class. The abstract class is one that used for only for inheritance i.e can not create the object of base class. The abstract class is one that contains at least one pure virtual function.
An abstract class is a class that is designed to be specifically used as a base class. You declare a pure virtual function by using a pure specifier (= 0) in the declaration of a virtual member function in the class declaration.

Abstract class is used in situation, when we have partial set of implementation of methods in a class. For example, consider a class has four methods. Out of four methods, we have an implementation of two methods and we need derived class to implement other two methods. In these kind of situations, we should use abstract class.

A virtual function will become pure virtual function when you append "=0" at the end of declaration of virtual function. A class with at least one pure virtual function or abstract function is called abstract class. Pure virtual function is also known as abstract function.

- We can't create an object of abstract class b'coz it has partial implementation of methods.
- Abstract function doesn't have body
- We must implement all abstract functions in derived class.

The following is an example of an abstract class: A Class

```
#include<iostream.h>
 #include<conio.h>
      class BaseClass       //Abstract class
      {     public:
              virtual void Display1( )=0;     //Pure virtual function or abstract function
              void Display3( )
              {     cout<<"\n\tThis is Display3() method of Base Class";
              }

      };
      class DerivedClass : public BaseClass
      {      public:
              void Display1()
              {
                    cout<<"\n\tThis is Display1() method of Derived Class";
              }
      };
     void main()
      {     DerivedClass D;
             D.Display1();            // This will invoke Display1() method of Derived Class
             D.Display3();            // This will invoke Display3() method of Base Class
      }
```

OVERLOADING

When there are more than one definition for a function name or an operator in the same scope, which is called function overloading and operator overloading respectively.

FUNCTION OVERLOADING

When multiple definitions for the same function name in the same scope is called “Function overloading”. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You can not overload function declarations that differ only by return type.

Example:

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

class printData
{
   public:
      void print(int i)
     {
        cout << "Printing int: " << i << endl;
      }

      void print(double  f)
     {
        cout << "Printing float: " << f << endl;
      }

      void print(char* c)
      {
        cout << "Printing character: " << c << endl;
      }
};

int main(void)
{
   printData pd;
   pd.print(5);
   pd.print(500.263);
   pd.print("Hello C++");
    return 0;
}
```

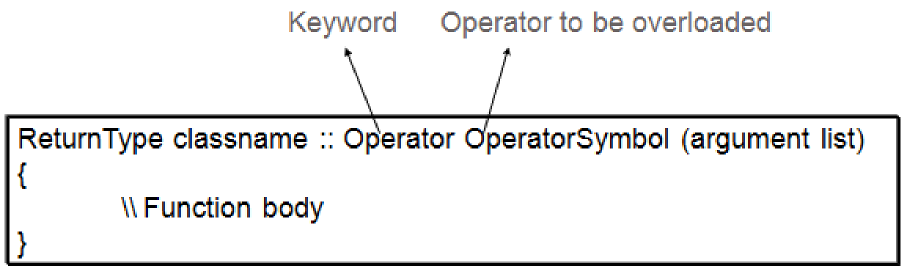
OPERATORS OVERLOADING

It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.

Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded. Operator that are not overloaded are follows

1. scope operator - ::
2. sizeof
3. member selector - .
4. member pointer selector - *
5. ternary operator - ?:

Operator Overloading Syntax



```
ReturnType classname :: Operator OperatorSymbol (argument list)
{
        \\ Function body
}
```

Example: To overload '+' Operator

```
#include< iostream.h>
class time
{   int h,m,s;
 public:
  time( )
  {  h=0, m=0; s=0;    }
  void getTime( );
  void show( )
  {   cout<< h<< ":"<< m<< ":"<< s;
  }
 time operator+(time);   //overloading '+' operator
};
time time::operator+(time t1) //operator function
```

```
{   time t;
 int a,b;
 a=s+t1.s;
 t.s=a%60;
 b=(a/60)+m+t1.m;
 t.m=b%60;
 t.h=(b/60)+h+t1.h;
 t.h=t.h%12;
 return t;
}
void time::getTime()
{  cout<<"\n Enter the hour(0-11) ";
  cin>>h;
  cout<<"\n Enter the minute(0-59) ";
  cin>>m;
  cout<<"\n Enter the second(0-59) ";
  cin>>s;
}
void main()
{   clrscr();
 time t1,t2,t3;
 cout<<"\n Enter the first time ";
 t1.getTime();
 cout<<"\n Enter the second time ";
 t2.getTime();
 t3=t1+t2;        //adding of two time object using '+' operator
 cout<<"\n First time ";
 t1.show();
 cout<<"\n Second time ";
 t2.show();
 cout<<"\n Sum of times ";
 t3.show();
 getch();
}
```

INITIALIZER LIST

Initializer List is used to initialize data members of a class. The list of members to be initialized is indicated with constructor as a comma separated list followed by a colon.

1. **For initialization of member objects which do not have default constructor:** In the following example, an object "a" of class "A" is data member of class "B", and "A" doesn't have default constructor. Initializer List must be used to initialize "a".

```
#include <iostream.h>
class A
{
   int i;
public:
   A(int );
};

A::A(int arg)
{
   i = arg;
   cout << "A's Constructor called: Value of i: " << i << endl;
}
// Class B contains object of A
class B
{
   A a;
public:
   B(int );
};

B::B(int x):a(x)
{  //Initializer list must be used
   cout << "B's Constructor called";
}

int main( )
 {
   B obj(10);
   return 0;
 }
/* OUTPUT:
   A's Constructor called: Value of i: 10
   B's Constructor called
*/
```

2. **When constructor's parameter name is same as data member** If constructor's parameter name is same as data member name then the data member

must be initialized either using this pointer or Initializer List. In the following example, both member name and parameter name for A() is “i”.

```
#include <iostream.h>
class A
{
     int i;
public:
   A(int );
    int getI( ) const { return i; }
};

A::A(int i):i(i) { }  // Either Initializer list or this pointer must be used
/* The above constructor can also be written as
A::A(int i) {
    this->i = i;
}
*/

int main( )
 {
  A a(10);
  cout<<a.getI();
  return 0;
}
```

OUTPUT: 10

3. **For initialization of non-static const data members:** const data members must be initialized using Initializer List. In the following example, “t” is a const data member of Test class and is initialized using Initializer List.

```
#include<iostream.h>
class Test
{  const int t;
public:
   Test(int t):t(t) {}                    //Initializer list must be used
   int getT( )
   {        return t;       }
};

int main( ) {
   Test t1(10);
   cout<<t1.getT();
   return 0;
}
```

OUTPUT: 10

FILE HANDLING

This feature of c++ enables you to handles files directly. This requires another standard C++ library called fstream, which defines three new data types:

Data Type	Description
Ofstream	This data type represents the output file stream and is used to create files and to write information to files.
Ifstream	This data type represents the input file stream and is used to read information from files.
Fstream	This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

FILE - A file is sequential stream of bytes ending with an end-of-file marker.

Types of file supported by C++:

- Text Files
- Binary Files

Difference between text file and binary file

- Text file is human readable because everything is stored in terms of text. In binary file everything is written in terms of 0 and 1, therefore binary file is not human readable.
- A newline(\n) character is converted into the carriage return-linefeed combination before being written to the disk. In binary file, these conversions will not take place.
- In text file, a special character, whose ASCII value is 26, is inserted after the last character in the file to mark the end of file. There is no such special character present in the binary mode files to mark the end of file.
- In text file, the text and characters are stored one character per byte. For example, the integer value 23718 will occupy 2 bytes in memory but it will occupy 5 bytes in text file. In binary file, the integer value 23718 will occupy 2 bytes in memory as well as in file.

OPENING A FILE

A file must be opened before you can read from it or write to it. Either the ofstream or fstream object may be used to open a file for writing and ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

```
void open(const char *filename, ios::openmode mode);
```

Here, the first argument specifies the name and location of the file to be opened and the second argument of the open() member function defines the mode in which the file should be opened.

Mode Flag	**Description**
ios::app	Append mode. All output to that file to be appended to the end.
ios::ate	Open a file for output and move the read/write control to the end of the file.
ios::in	Open a file for reading.
ios::out	Open a file for writing.
ios::trunc	If the file already exists, its contents will be truncated before opening the file.

CLOSING A FILE

When a C++ program terminates it automatically closes flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

WRITING TO A FILE

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an ofstream or fstream object instead of the cout object.

READING FROM A FILE

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an ifstream or fstream object instead of the cin object.

EXAMPLE:

```
#include <fstream>
#include <iostream>
int main ()
{
   char data[100];
  ofstream outfile;
  outfile.open("afile.dat");
  cout << "Writing to the file" << endl;
  cout << "Enter your name: ";
  cin.getline(data, 100);
  outfile << data << endl;                  // write inputted data into the file.
  cout << "Enter your age: ";
  cin >> data;
  cin.ignore( );
  outfile << data << endl;                  // again write inputted data into the file.
  outfile.close( );                         // close the opened file.
  ifstream infile;                          // open a file in read mode.
  infile.open("afile.dat");
  cout << "Reading from the file" << endl;
  infile >> data;
  cout << data << endl;                     // write the data at the screen.
  infile >> data;                          // again read the data from the file and display it.
  cout << data << endl;
  infile.close();                          // close the opened file.
  return 0;
}
```

WRITE() FUNCTION

The write() function is used to write object or record (sequence of bytes) to the file. A record may be an array, structure or class.

Syntax of write() function

```
fstream fout;

fout.write( (char *) &obj, sizeof(obj) );
```

The write() function takes two arguments.
&obj : Initial byte of an object stored in memory.
sizeof(obj) : size of object represents the total number of bytes to be written from initial byte
Example:

```
#include<fstream.h>
#include<conio.h>
class Student
{
        int roll;
        char name[25];
        float marks;
        void getdata()
        {
                cout<<"\n\nEnter Roll : ";
                cin>>roll;
                cout<<"\nEnter Name : ";
                cin>>name;
                cout<<"\nEnter Marks : ";
                cin>>marks;
        }

        public:
        void AddRecord()
        {
                fstream f;
                Student Stu;
                f.open("Student.dat",ios::app|ios::binary);
                Stu.getdata();
                f.write( (char *) &Stu, sizeof(Stu) );
                f.close( );
        }
};

void main( )
{
        Student S;
        char ch='n';
        do
        {
                S.AddRecord( );
                cout<<"\n\nDo you want to add another data (y/n) : ";
                ch = getche( );
        } while(ch=='y' || ch=='Y');
        cout<<"\nData written successfully...";
}
```

READ( ) FUNCTION

The read() function is used to read object (sequence of bytes) to the file.

Syntax of read() function

```
 fstream fin;
fin.read( (char *) &obj, sizeof(obj) );
```

The read() function takes two arguments.
&obj : Initial byte of an object stored in file.
sizeof(obj) : size of object represents the total number of bytes to be read from initial byte.

The read() function returns NULL if no data read.

Example:

```
#include<fstream.h>
 #include<conio.h>
 class Student
 {
        int roll;
        char name[25];
        float marks;
        void putdata( )
        {
           cout<<"\n\t"<<roll<<"\t"<<name<<"\t"<<marks;
        }
        public:
        void Display( )
        {
               fstream f;
               Student Stu;
               f.open("Student.dat",ios::in|ios::binary);
               cout<<"\n\tRoll\tName\tMarks\n";
               while( (f.read((char*)&Stu,sizeof(Stu))) != NULL )
                              Stu.putdata();
               f.close();
        }
 };

 void main()
 {
        Student S;
        S.Display();
 }
```

FILE POSITION POINTERS

Both istream and ostream provide member functions for repositioning the file-position pointer. These member functions are seekg ("seek get") for istream and seekp ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be ios::beg (the default) for positioning relative to the beginning of a stream, ios::cur for positioning relative to the current position in a stream or ios::end for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are:

```
fileObject.seekg( n );            // position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n, ios::cur );  // position n bytes forward in fileObject
fileObject.seekg( n, ios::end );  // position n bytes back from end of fileObject
fileObject.seekg( 0, ios::end );  // position at end of fileObject
```

EXAMPLE:

```
#include <fstream.h>
int main()
{
   long position;
   fstream file;
   file.open("myfile.txt");
   file.write("this is an apple", 16);
   position = file.tellp( );
   file.seekp(position - 7);
   file.write(" sam", 4);
   file.close();
}
```

Output –

```
this is a sample
```

Here tellp( ) function returns the position of pointer then using seekp() function the pointer is shift back from n position here it shift 7 position back and then insert the content at that position .

Program to count number of characters.

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

int main()
{
   ifstream fin;
   fin.open("out.txt");
   int count = 0;
   char ch;
   while(!fin.eof())
   {
      fin.get(ch);
      count++;
   }
   cout << "Number of characters in file are " << count;
   fin.close();
   return 0;
}
```

Program to copy contents of file to another file.

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

int main()
{
   ifstream fin;
   fin.open("out.txt");
   ofstream fout;
   fout.open("sample.txt");
   char ch;
   while(!fin.eof())
   {
      fin.get(ch);
      fout << ch;
   }
   fin.close();
   fout.close();
   return 0;
}
```

Program to count number of lines

```
#include<fstream.h>
#include<iostream.h>
int main()
```

```
{
   ifstream fin;
   fin.open("out.txt");
   int count = 0;
   char str[80];
   while(!fin.eof())
   {
      fin.getline(str,80);
      count++;
   }
   cout << "Number of lines in file are " << count;
   fin.close();
   return 0;
}
```

BASIC OPERATION ON BINARY FILE IN C++

When data is stored in a file in the binary format, reading and writing data is faster because no time is lost in converting the data from one format to another format. Such files are called binary files. This following program explains how to create binary files and also how to read, write, search, delete and modify data from binary files.

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

class Student
{
   int admno;
   char name[50];
public:
   void setData()
   {
      cout << "\nEnter admission no. ";
      cin >> admno;
      cout << "Enter name of student ";
      cin.getline(name,50);
   }

   void showData()
   {
      cout << "\nAdmission no. : " << admno;
      cout << "\nStudent Name : " << name;
   }

   int retAdmno()
   {
      return admno;
```

```
    }
};


void write_record( )
{
    ofstream outFile;
    outFile.open("student.dat", ios::binary | ios::app);
    Student obj;
    obj.setData( );
    outFile.write((char*)&obj, sizeof(obj));
    outFile.close();
}



void display( )
{
    ifstream inFile;
    inFile.open("student.dat", ios::binary);
    Student obj;
    while(inFile.read((char*)&obj, sizeof(obj)))
    {
        obj.showData();
    }
    inFile.close();
}

void search(int n)
{
    ifstream inFile;
    inFile.open("student.dat", ios::binary);
    Student obj;
    while(inFile.read((char*)&obj, sizeof(obj)))
    {
        if(obj.retAdmno() == n)
        {
            obj.showData();
        }
    }
    inFile.close();
}

void delete_record(int n)
{
    Student obj;
    ifstream inFile;
    inFile.open("student.dat", ios::binary);
    ofstream outFile;
    outFile.open("temp.dat", ios::out | ios::binary);
    while(inFile.read((char*)&obj, sizeof(obj)))
    {
        if(obj.retAdmno() != n)
        {
            outFile.write((char*)&obj, sizeof(obj));
        }
    }
    inFile.close();
    outFile.close();
    remove("student.dat");
    rename("temp.dat", "student.dat");
}

void modify_record(int n)
{
    fstream file;
    file.open("student.dat",ios::in | ios::out);
    Student obj;
    while(file.read((char*)&obj, sizeof(obj)))
    {
        if(obj.retAdmno() == n)
        {
            cout << "\nEnter the new details of student";
            obj.setData( );
            int pos = -1 * sizeof(obj);
            file.seekp(pos, ios::cur);
            file.write((char*)&obj, sizeof(obj));
        }
    }
    file.close();
}

int main()
{
    for(int i = 1; i <= 4; i++)
    { write_record( );                              //Store 4 records in file
    }
    cout << "\nList of records"                 //Display all records
    display( );
    cout << "\nSearch result";                  //Search record
    search(100);
    delete_record(100);                         //Delete record
    cout << "\nRecord Deleted";
    cout << "\nModify Record 101 ";             //Modify record
    modify_record(101);
    return 0;
}
```

THIS PONTER

C++ provides a keyword 'this', which represents the current object and passed as a hidden argument to all member functions.

The **this** pointer is a constant pointer that holds the memory address of the current object. The **this** pointer is not available in static member functions as static member functions can be called without any object. static member functions can be called with class name.

```
#include<iostream.h>
class Test
{private:
  int x;
public:
  void setX (int x)
  {           this->x = x;
  }
  void print() { cout << "x = " << x << endl; }
};
void main()
{
  Test obj;
  int x = 20;
  obj.setX(x);
  obj.print();
}
```

Demonstration

```
#include <iostream.h>
class MyClass {
      int data;
   public:
      MyClass() {data=100;};
      void Print1();
      void Print2();
};
// Not using this pointer
void MyClass::Print1() {
   cout << data << endl;
}
// Using this pointer
void MyClass::Print2() {
   cout << "My address = " << this << endl;
   cout << this->data << endl;
}
```

```
int main()
{
   MyClass a;
   a.Print1();
   a.Print2();
   // Size of doesn't include this pointer
   cout << sizeof(a) << endl;
}
```

OUTPUT:

```
100
My address = 0012FF88
100
4
```

STATIC KEYWORD

Static is a keyword in C++ used to give special characteristics to an element. Static elements are allocated storage only once in a program lifetime in static storage area. And they have a scope till the program lifetime. Static Keyword can be used with following,

1. Static variable in functions
2. Static Class Objects
3. Static member Variable in class
4. Static Methods in class

STATIC VARIABLES INSIDE FUNCTIONS

Static variables when used inside function are initialized only once, and then they hold there value even through function calls.

These static variables are stored on static storage area , not in stack.

```
void counter( )
{
 static int count=0;
 cout << count++;
}

int main( )
{
 for(int i=0;i<5;i++)
 {
   counter( );
 }
}
```

Output : 0 1 2 3 4

If we do not use static keyword, the variable count, is reinitialized everytime when counter() function is called, and gets destroyed each time when counter() functions ends. But, if we make it static, once initialized count will have a scope till the end of main() function and it will carry its value through function calls too.
If you don't initialize a static variable, they are by default initialized to zero.

STATIC CLASS OBJECTS

Static keyword works in the same way for class objects too. Objects declared static are allocated storage in static storage area, and have scope till the end of program.

Static objects are also initialized usig constructors like other normal objects. Assignment to zero, on using static keyword is only for primitive datatypes, not for user defined datatypes.

```
class Abc
{ int i;
 public:
 Abc()
 {
  i=0;
  cout << "constructor";
 }
 ~Abc( )
 {   cout << "destructor";
 }
};

void f( )
{
 static Abc obj;
}

int main( )
{ int x=0;
 if(x==0)
 {
  f( );
 }
 cout << "END";
}
```

Output : constructor END destructor

The destructor not called upon the end of the scope of if condition. This is because object was static, which has scope till the program lifetime, hence destructor for this object was called when main() exits.

STATIC DATA MEMBER IN CLASS

Static data members of class are those members which are shared by all the objects. Static data member has a single piece of storage, and is not available as separate copy with each object, like other non-static data members. Static member variables (data members) are not initialied using constructor, because these are not dependent on object initialization.

Also, it must be initialized explicitly, always outside the class. If not initialized, Linker will give error.

```
class X
```

```
{
 static int i;
 public:
 X( ){};
};

int X::i=1;

int main( )
{
 X obj;
 cout << obj.i;   // prints value of i
}
```

Once the definition for static data member is made, user cannot redefine it. Though, arithmetic operations can be performed on it.

STATIC MEMBER FUNCTIONS

These functions work for the class as whole rather than for a particular object of a class.

It can be called using an object and the direct member access . operator. But, its more typical to call a static member function by itself, using class name and scope resolution :: operator.

Example :

```
class X
{
 public:
 static void f( ){};
};

int main( )
{
 X::f( );   // calling member function directly with class name
}
```

These functions cannot access ordinary data members and member functions, but only static data members and static member functions.

EXCEPTION HANDLING

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

- **throw:** A program throws an exception when a problem shows up. This is done using a throw keyword.
- **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **try:** A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Syntax:

```
try
{
   // protected code
}
catch( ExceptionName e1 )
{
   // catch block
}
```

Example:

```
 #include <iostream.h>
double division(int a, int b)
{
   if( b == 0 )
   {
      throw "Division by zero condition!";
   }
   return (a/b);
}
int main ()
{
   int x = 50;
   int y = 0;
   double z = 0;
   try {
     z = division(x, y);
     cout << z << endl;
   }
  catch (char msg[200])
   {
     cerr << msg << endl;
   }
   return 0;
}
```

MULTIPLE CATCH

Mutilple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Syntax:

```
try
{
  // protected code
}
catch( ExceptionName e1 )
{
  // catch block
}
catch( ExceptionName e2 )
{
  // catch block
}
catch( ExceptionName eN )
{
  // catch block
}
```

Example:

```
    #include<iostream.h>
     #include<conio.h>
     void main()
     {
         int a=2;

           try
           {

              if(a==1)
                  throw a;                  //throwing integer exception

              else if(a==2)
                  throw 'A';                 //throwing character exception

              else if(a==3)
                  throw 4.5;                 //throwing float exception

           }
```

```
        catch(int a)
        {
           cout<<"\nInteger exception caught.";
        }
        catch(char ch)
        {
           cout<<"\nCharacter exception caught.";
        }
        catch(double d)
        {
           cout<<"\nDouble exception caught.";
        }

        cout<<"\nEnd of program.";

    }
```

Output :

```
        Character exception caught.
        End of program.
```

CATCH ALL EXCEPTIONS

The above example will caught only three types of exceptions that are integer, character and double. If an exception occur of long type, no catch block will get execute and abnormal program termination will occur. To avoid this, We can use the catch statement with three dots as parameter (...) so that it can handle all types of exceptions.

Example :

```
    #include<iostream.h>
    #include<conio.h>
    void main()
    {
       int a=1;
        try
        {

           if(a==1)
              throw a;              //throwing integer exception

           else if(a==2)
              throw 'A';            //throwing character exception

           else if(a==3)
              throw 4.5;            //throwing float exception

        }
        catch(...)
        {
            cout<<"\nException occur.";
        }

        cout<<"\nEnd of program.";
    }
```

Output :

```
        Exception occur.
        End of program
```

<u>NESTED TRY/ RETHROWING EXCEPTIONS</u>

Rethrowing exception is possible, where we have an inner and outer try-catch statements (Nested try-catch). An exception to be thrown from inner catch block to outer catch block is called rethrowing exception.

```
    #include<iostream.h>
    #include<conio.h>
    void main( )
    {   int a=1;
        try
        {
            try
            {
                throw a;
            }
            catch(int x)
            {
              cout<<"\nException in inner try-catch block.";
              throw x;
            }
        }
        catch(int n)
        {  cout<<"\nException in outer try-catch block.";
        }
        cout<<"\nEnd of program.";
    }
```

Output :

```
        Exception in inner try-catch block.
        Exception in outer try-catch block.
        End of program.
```

<u>DYNAMIC MEMORY ALLOCATION</u>

Dynamic memory allocation means creating memory at runtime. For example, when we declare an array, we must provide size of array in our source code to allocate memory at compile time.

But if we need to allocate memory at runtime me must use new operator followed by data type. If we need to allocate memory for more than one element, we must provide total number of elements required in square bracket[ ]. It will return the address of first byte of memory.

NEW OPERATOR

The new operator is used to allocate memory dynamically.

Syntax: new data-type;

Example:

```
double* pvalue  = NULL;          // Pointer initialized with null
```

```
pvalue  = new double;                      // Request memory for the variable
```

DELETE OPERATOR

The delete operator is used to deallocate memory dynamically which is given by new.

```
#include <iostream.h>
int main ()
{
  double* pvalue  = NULL;                          // Pointer initialized with null
  pvalue  = new double;                            // Request memory for the variable
  *pvalue = 29494.99;                              // Store value at allocated address
  cout << "Value of pvalue : " << *pvalue << endl;
  delete pvalue;                                   // free up the memory.
  return 0;
}
```

Example:

```
#include<iostream.h>
#include<conio.h>
          void main()
          {

                    int size,i;
                    int *ptr;
                    cout<<"\n\tEnter size of Array : ";
                    cin>>size;
                    ptr = new int[size];
                    //Creating memory at run-time and return first byte of address to ptr.

                    for(i=0;i<5;i++)       //Input arrray from user.
                    {
                     cout<<"\nEnter any number : ";
                     cin>>ptr[i];
```

```
                }

                 for(i=0;i<5;i++)         //Output arrray to console.
                {
                  cout<<ptr[i]<<", ";
                }

                 delete[] ptr;              //deallocating all the memory created by new
        }
```

Output :

```
                Enter size of Array : 5

                Enter any number : 78
                Enter any number : 45
                Enter any number : 12
                Enter any number : 89
                Enter any number : 56

                78, 45, 12, 89, 56
```

<u>DYNAMIC MEMORY ALLOCATION FOR OBJECTS</u>

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept:

```
#include <iostream.h>
class Box
{   public:
      Box( )
       {
          cout << "Constructor called!" <<endl;
        }
      ~Box( )
        {
          cout << "Destructor called!" <<endl;
         }
};
int main( )
{
   Box* myBoxArray = new Box[4];
   delete[ ] myBoxArray;                              // Delete array
   return 0;
}
```

REFERENCES

References are like constant pointers that are automatically dereferenced. It is a new name given to an existing storage. So when you are accessing the reference, you are actually accessing that storage.

```
int main()
{ int y=10;
  int &r = y;  // r is a reference to int y
  cout << r;
}
```

Difference between Reference and Pointer

References	Pointers
Reference must be initialized when it is created.	Pointers can be initialized any time.
Once initialized, we cannot reinitialize a reference.	Pointers can be reinitialized any number of time.
You can never have a NULL reference.	Pointers can be NULL.
Reference is automatically dereferenced.	* is used to dereference a pointer

CONST REFERENCE

Const reference is used in function arguments to prevent the function from changing the argument.

```
void g(const int& x)
{ x++;
}   // ERROR

int main()
{
 int i=10;
 g(i);
}
```

We cannot change the argument in the function because it is passed as const reference.

STRINGS

C++ provides following two types of string representations:

- The C-style character string.
- The string class type introduced with Standard C++.

THE C-STYLE CHARACTER STRING

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.

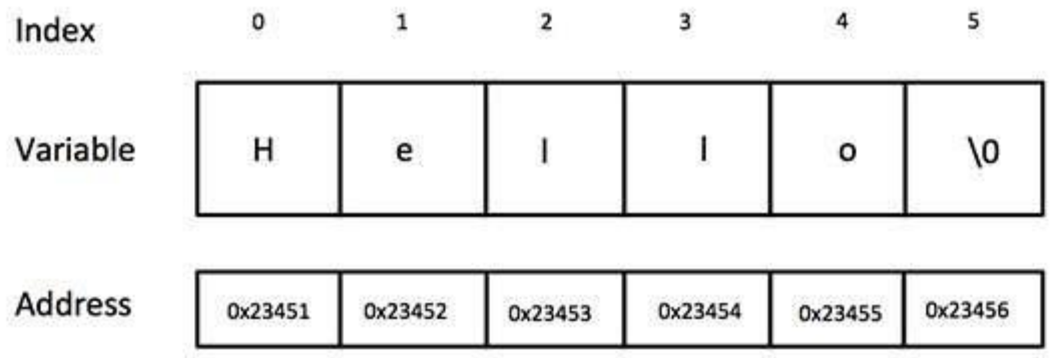
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization, then you can write the above statement as follows:

```
char greeting[] = "Hello";
```

Following is the memory presentation of above defined string in C/C++:



Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string:

```
#include <iostream.h>
int main ()
{
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
   cout << "Greeting message: ";
   cout << greeting << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Greeting message: Hello
```

Reading strings with/without embedded blanks

To read a string without blanks cin can be used

```
cin>>str;
```

To read a string with blanks cin.getline() or gets() can be used.

```
cin.getline(str,80);
```

-Or-

```
gets(str);
```

C++ supports a wide range of functions that manipulate null-terminated strings:

S.N.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

Following example makes use of few of the above-mentioned functions:

```
#include <iostream.h>
#include <cstring.h>
int main ()
{ char str1[10] = "Hello";
  char str2[10] = "World";
  char str3[10];
  int  len ;
  // copy str1 into str3
  strcpy( str3, str1);
  cout << "strcpy( str3, str1) : " << str3 << endl;
  // concatenates str1 and str2
  strcat( str1, str2);
  cout << "strcat( str1, str2): " << str1 << endl;
  // total lenghth of str1 after concatenation
  len = strlen(str1);
  cout << "strlen(str1) : " << len << endl;
  return 0;
}
```

STRING HEADER FILE

The standard C++ library provides a string class type that supports all the operations mentioned above, additionally much more functionality. We will study this class in C++ Standard Library but for now let us check following example:

At this point, you may not understand this example because so far we have not discussed Classes and Objects. So can have a look and proceed until you have understanding on Object Oriented Concepts.

```
#include <iostream.h>
#include <string.h>
int main ()
{
  string str1 = "Hello";
  string str2 = "World";
  string str3;
  int  len ;
  // copy str1 into str3
  str3 = str1;
  cout << "str3 : " << str3 << endl;
  // concatenates str1 and str2
  str3 = str1 + str2;
  cout << "str1 + str2 : " << str3 << endl;
  // total lenghth of str3 after concatenation
  len = str3.size();
  cout << "str3.size() :  " << len << endl;
  return 0;
}
```

Function to count the number of words in a string

```
void count(char S[])
{  int words=0;
  for(int i=0;S[i]!='\0';i++)
  {  if (S[i]==' ')
      words++;              //Checking for spaces
  }
  cout<<"The number of words="<<words+1<<endl;
}
```

ASSOCIATION

Association is a simple structural connection or channel between classes and is a relationship where all objects have their own lifecycle and there is no owner.
There can be two type of associations. They are-

1. Composition
2. Aggregation

COMPOSITION

Composition is again specialize form of Aggregation. It is a strong type of Aggregation. Here the Parent and Child objects have coincident lifetimes. Child object does not have it's own lifecycle and if parent object gets deleted, then all of it's child objects will also be deleted.

```
Class A
{  public:
     A();
     ~A();
};
Class B
{   public:
      B();
    ~B();
   private:
       A a;//A composition to B
};
```

AGGREGATION

Aggregation is a specialize form of Association where all object have their own lifecycle but there is a ownership like parent and child. Child object can not belong to another parent object at the same time. We can think of it as "has-a" relationship. If a parent child is deleted, objects of child classes will not be deleted.
Example:

```
Class A
{     public:
        A();
       ~A();
};
 Class B
{     public:
        B();
       ~B();
        SayHello()
        {             a = new A();
        }
     private:
        A *a;//A aggregation to B
};
```

Example: **To demonstrate aggregation and composition**

```
#include<iostream.h>
class B
{public:
   B(int id)
   {       this->ID = id;
      cout<<"construct a B object: "<<id<<endl;
   }
  ~B()
   {       cout<<"destruct the B class object: "<<this->ID<<endl;
   }
private:
   int ID;
};
class C
{public:
   C()
   {  cout<<"construct C object"<<endl;
   }
   ~C()
   {
      cout<<"destruct C object"<<endl;
   }
private:
};
class A
{public:
   A()
   {       cout<<"construct A object"<<endl;    }
   ~A()
   {       cout<<"destruct A object"<<endl;
   }
    void Aggregation()
   {       bPtr = new B(1);
   }
private:
   B* bPtr;                 //aggregation
   C composition;       //composition
};
void main()
{    cout<<"Demo for class aggregation relationship"<<endl;
   A a;
   a.Aggregation();

}
```

CONST KEYWORD

Constant is something that doesn't change. In C and C++ we use the keyword const to make program elements constant. Const keyword can be used in many context in a C++ program. Const keyword can be used with:

1. Variables
2. Pointers
3. Function arguments and return types
4. Class Data members
5. Class Member functions
6. Objects

1) CONSTANT VARIABLES

If you make any variable as constant, using const keyword, you cannot change its value. Also, the constant variables must be initialized while declared.

```
int main
{
 const int i = 10;
 const int j = i+10;  // Works fine
 i++;    // This leads to Compile time error
}
```

In this program we have made **i** as constant, hence if we try to change its value, compile time error is given. Though we can use it for substitution.

2) POINTERS WITH CONST

Pointers can be made **const** too. When we use const with pointers, we can do it in two ways, either we can apply const to what the pointer is pointing to, or we can make the pointer itself a const.

Pointer to Const

This means that the pointer is pointing to a const variable.

```
const int* u;
```

Here, u is a pointer that points to a **const int**. We can also write it like,

```
int const* v;
```

still it has the same meaning. In this case also, v is a pointer to an int which is const.

Const pointer

To make the pointer const, we have to put the **const** keyword to the right of the `*`.

```
int x = 1;
int* const w = &x;
```

Here, w is a pointer, which is const, that points to an int. Now we can't change the pointer but can change the value that it points to.

NOTE : We can also have a const pointer pointing to a const variable.

```
const int* const x;
```

3) CONST FUNCTION ARGUMENTS AND RETURN TYPES

We can make the return type or arguments of a function as const. Then we cannot change any of them.

```
void f(const int i)
{
 i++;    // Error
}

const int g()
{
 return 1;
}
```

Some Important points to remember

1. For built in types, returning a const or non-const, doesn't amke any difference.

```
const int h()
{
 return 1;
}
it main(0
{
 const int j = h();
 int k = h();
}
```

Both j and k will be assigned 1. No error will occur.

For user defined data types, returning const, will prevent its modification.

2. Temporary objects created while program execution are always of const type.
3. If a function has a non-const parameter, it cannot be passed a const argument while making a call.

   ```
   void t(int*) { }
   ```

If we pass a const int* argument, it will give error.

4. But, a function which has a const type parameter, can be passed a const type argument as well as a non-const argument.

   void g(const int*) {}

   This function can have a int* as well as const int* type argument.

4) <u>CONST CLASS DATA MEMBERS</u>

These are data variables in class which are made const. They are not initialized during declaration. Their initialization occur in the constructor.

```
class Test
{ const int i;
 public:
 Test (int x)
 {   i=x;
 }
};
int main()
{
 Test t(10);
 Test s(20);
}
```

In this program, i is a const data member, in every object its independent copy is present, hence it is initialized with each object using constructor. Once initialized, it cannot be changed.

5) <u>CONST CLASS OBJECT</u>

When an object is declared or created with const, its data members can never be changed, during object's lifetime.

Syntax :
const class_name object;

Const class Member function

A const member function never modifies data members in an object.

Syntax :
return_type function_name() const;

Example for const Object and const Member function

class X

```
{
 int i;
 public:
 X(int x)   // Constructor
 { i=x; }
 int f() const    // Constant function
 { i++; }

 int g()
 { i++; }
};
int main()
{
X obj1(10);          // Non const Object
const X obj2(20);   // Const Object
obj1.f();   // No error
obj2.f();   // No error
cout << obj1.i << obj2.i ;
obj1.g();   // No error
obj2.g();   // Compile time error
}
Output : 10 20
```

Here, we can see, that const member function never changes data members of class, and it can be used with both const and non-const object. But a const object can't be used with a member function which tries to change its data members.

MUTABLE KEYWORD

Mutable keyword is used with member variables of class, which we want to change even if the object is of const type. Hence, mutable data members of a const objects can be modified.

```
class Z
{ int i;
 mutable int j;
 public:
 Z()
 {i=0; j=0;}
 void f() const
 { i++;  // Error
   j++;  // Works, because j is Mutable
 }
};
int main(0
{
 const Z obj;
 obj.f();
}
```

TEMPLATES

A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

Templates can be represented in two ways:

- Function templates
- Class templates

Function Templates-We can define a template for a function. For example, if we have an add( ) function, we can create versions of the add function for adding the int, float or double type values.

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

Syntax of Function Template

```
template < class Ttype> ret_type func_name(parameter_list)
{
  // body of function.
}
```

Where **Ttype:** It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

class: A class keyword is used to specify a generic type in a template declaration.

```
#include <iostream.h>

template<class T> T add(T &a,T &b)
{
   T result = a+b;
   return result;

}
int main()
{
  int i =2;
  int j =3;
  float m = 2.3;
  float n = 1.2;
  cout<<"Addition of i and j is :"<<add(i,j);
  cout<<'\n';
  cout<<"Addition of m and n is :"<<add(m,n);
  return 0;
}
```

CLASS TEMPLATE-Class Template can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.

Syntax

```
template<class Ttype>
class class_name
{
  .
  .
}
```

Ttype is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

```
#include <iostream.h>
template<class T>
class A
{
   public:
   T num1 = 5;
   T num2 = 6;
   void add()
   {
      cout << "Addition of num1 and num2 : " << num1+num2<< endl;
   }

};

int main()
{
   A<int> d;
   d.add( );
   return 0 ;
}
```

NAMESPACES

Namespaces in C++ are used to organize too many classes so that it can be easy to handle the application. For accessing the class of a namespace, we need to use namespacename::classname. We can use **using** keyword so that we don't have to use complete name all the time.

In C++, global namespace is the root namespace. The global::std will always refer to the namespace "std" of C++ Framework.

Example:

```
#include <iostream>
using namespace std;
namespace First
{
   void sayHello( )
   {
      cout<<"Hello First Namespace"<<endl;
   }
}
namespace Second
{
     void sayHello( )
     {
        cout<<"Hello Second Namespace"<<endl;
     }
}
int main()
{
 First::sayHello();
 Second::sayHello();
return 0;
}
```

PROJECT: SUPERMARKET BILLING SYSTEM

```
#include<conio.h>
#include<stdio.h>
#include<process.h>
#include<fstream.h>
//***************************************************************
// CLASS USED IN PROJECT
//****************************************************************
class product
{
int pno;
char name[50];
float price,qty,tax,dis;
public:
void create_product( )
{
cout<<"\nPlease Enter The Product No. of The Product ";
cin>>pno;
cout<<"\n\nPlease Enter The Name of The Product ";
gets(name);
cout<<"\nPlease Enter The Price of The Product ";
cin>>price;
cout<<"\nPlease Enter The Discount (%) ";
cin>>dis;
}
void show_product( )
{
cout<<"\nThe Product No. of The Product : "<<pno;
cout<<"\nThe Name of The Product : ";
puts(name);
cout<<"\nThe Price of The Product : "<<price;
cout<<"\nDiscount : "<<dis;
}
int retpno( )
{return pno;}
float retprice( )
{return price;}
char* retname( )
{return name;}
int retdis( )
{return dis;}
}; //class ends here

//***************************************************************
// global declaration for stream object, object
//****************************************************************
fstream fp;
product pr;
```

```
//************************************************************
// function to write in file
//*************************************************************
void write_product( )
{
fp.open("Shop.dat",ios::out|ios::app);
pr.create_product( );
fp.write((char*)&pr,sizeof(product));
fp.close( );
cout<<"\n\nThe Product Has Been Created ";
getch( );
}
//************************************************************
// function to read all records from file
//*************************************************************
void display_all( )
{
clrscr( );
cout<<"\n\n\n\t\tDISPLAY ALL RECORD !!!\n\n";
fp.open("Shop.dat",ios::in);
while(fp.read((char*)&pr,sizeof(product)))
{
pr.show_product( );
cout<<"\n\n====================================\n";
getch();
}
fp.close( );
getch( );
}
//************************************************************
// function to read specific record from file
//*************************************************************
void display_sp(int n)
{
int flag=0;
fp.open("Shop.dat",ios::in);
while(fp.read((char*)&pr,sizeof(product)))
{
if(pr.retpno( )==n)
{
clrscr( );
pr.show_product();
flag=1;
}
}
fp.close( );
if(flag==0)
cout<<"\n\nrecord not exist";
```

```
getch( );
}
//***************************************************************
// function to modify record of file
//****************************************************************
void modify_product( )
{
int no,found=0;
clrscr( );
cout<<"\n\n\tTo Modify ";
cout<<"\n\n\tPlease Enter The Product No. of The Product";
cin>>no;
fp.open("Shop.dat",ios::in|ios::out);
while(fp.read((char*)&pr,sizeof(product)) && found==0)
{
if(pr.retpno( )==no)
{
pr.show_product( );
cout<<"\nPlease Enter The New Details of Product"<<endl;
pr.create_product( );
int pos=-1*sizeof(pr);
fp.seekp(pos,ios::cur);
fp.write((char*)&pr,sizeof(product));
cout<<"\n\n\t Record Updated";
found=1;
}
}
fp.close( );
if(found==0)
cout<<"\n\n Record Not Found ";
getch( );
}
//***************************************************************
// function to delete record of file
//****************************************************************
void delete_product( )
{
int no;
clrscr( );
cout<<"\n\n\n\tDelete Record";
cout<<"\n\nPlease Enter The product no. of The Product You Want To Delete";
cin>>no;
fp.open("Shop.dat",ios::in|ios::out);
fstream fp2;
fp2.open("Temp.dat",ios::out);
fp.seekg(0,ios::beg);
while(fp.read((char*)&pr,sizeof(product)))
{
if(pr.retpno( )!=no)
{
fp2.write((char*)&pr,sizeof(product));
}
}
fp2.close( );
fp.close( );
remove("Shop.dat");
rename("Temp.dat","Shop.dat");
cout<<"\n\n\tRecord Deleted ..";
getch();
}
//***************************************************************
// function to display all products price list
//****************************************************************
void menu( )
{
clrscr( );
fp.open("Shop.dat",ios::in);
if(!fp)
{
cout<<"ERROR!!! FILE COULD NOT BE OPEN\n\n\n Go To Admin Menu to create
File";
cout<<"\n\n\n Program is closing ....";
getch( );
exit(0);
}
cout<<"\n\n\t\tProduct MENU\n\n";
cout<<"====================================================\n";
cout<<"P.NO.\t\tNAME\t\tPRICE\n";
cout<<"====================================================\n";
while(fp.read((char*)&pr,sizeof(product)))
{
cout<<pr.retpno( )<<"\t\t"<<pr.retname()<<"\t\t"<<pr.retprice( )<<endl;
}
fp.close( );
}

//***************************************************************
// function to place order and generating bill for Products
//****************************************************************
void place_order( )
{
int order_arr[50],quan[50],c=0;
float amt,damt,total=0;
char ch='Y';
menu( );
cout<<"\n============================";
cout<<"\n PLACE YOUR ORDER";
cout<<"\n============================\n";
```

```
do{
cout<<"\n\nEnter The Product No. Of The Product : ";
cin>>order_arr[c];
cout<<"\nQuantity in number : ";
cin>>quan[c];
c++;
cout<<"\nDo You Want To Order Another Product ? (y/n)";
cin>>ch;
}while(ch=='y' ||ch=='Y');
cout<<"\n\nThank You For Placing The Order";getch();clrscr();
cout<<"\n
\n*******************************INVOICE************************\n";
cout<<"\nPr No.\tPr Name\tQuantity \tPrice \tAmount \tAmount after
discount\n";
for(int x=0;x<=c;x++)
{
fp.open("Shop.dat",ios::in);
fp.read((char*)&pr,sizeof(product));
while(!fp.eof( ))
{
if(pr.retpno( )==order_arr[x])
{
amt=pr.retprice( )*quan[x];
damt=amt-(amt*pr.retdis()/100);
cout<<"\n"<<order_arr[x]<<"\t"<<pr.retname()
<<"\t"<<quan[x]<<"\t\t"<<pr.retprice()<<"\t"<<amt<<"\t\t"<<damt;
total+=damt;
}
fp.read((char*)&pr,sizeof(product));
}
fp.close( );
}
cout<<"\n\n\t\t\t\t\tTOTAL = "<<total;
getch( );
}
//***************************************************************
// INTRODUCTION FUNCTION
//****************************************************************
void intro( )
{
clrscr( );
gotoxy(31,11);
cout<<"SUPER MARKET";
gotoxy(35,14);
cout<<"BILLING";
gotoxy(35,17);
cout<<"PROJECT";
cout<<"\n\nMADE BY : ANUJ KUMAR";
```

```
cout<<"\n\nSCHOOL : RYAN INTERNATIONAL SCHOOL";
getch();
}

//***************************************************************
// ADMINSTRATOR MENU FUNCTION
//****************************************************************
void admin_menu( )
{
clrscr( );
char ch2;
cout<<"\n\n\n\tADMIN MENU";
cout<<"\n\n\t1.CREATE PRODUCT";
cout<<"\n\n\t2.DISPLAY ALL PRODUCTS";
cout<<"\n\n\t3.QUERY ";
cout<<"\n\n\t4.MODIFY PRODUCT";
cout<<"\n\n\t5.DELETE PRODUCT";
cout<<"\n\n\t6.VIEW PRODUCT MENU";
cout<<"\n\n\t7.BACK TO MAIN MENU";
cout<<"\n\n\tPlease Enter Your Choice (1-7) ";
ch2=getche( );
switch(ch2)
{
case '1': clrscr( );
write_product( );
break;
case '2': display_all();break;
case '3':
int num;
clrscr( );
cout<<"\n\n\tPlease Enter The Product No. ";
cin>>num;
display_sp(num);
break;
case '4': modify_product();break;
case '5': delete_product();break;
case '6': menu();
getch();
case '7': break;
default:cout<<"\a";admin_menu();
}
}
//***************************************************************
// THE MAIN FUNCTION OF PROGRAM
//****************************************************************
void main( )
{
char ch;
intro( );
```

```
do
{
clrscr( );
cout<<"\n\n\n\tMAIN MENU";
cout<<"\n\n\t01. CUSTOMER";
cout<<"\n\n\t02. ADMINISTRATOR";
cout<<"\n\n\t03. EXIT";
cout<<"\n\n\tPlease Select Your Option (1-3) ";
ch=getche( );
switch(ch)
{
case '1': clrscr();
place_order();
getch( );
break;
case '2': admin_menu();
break;
case '3':exit(0);
default :cout<<"\a";
}
}while(ch!='3');
}
```