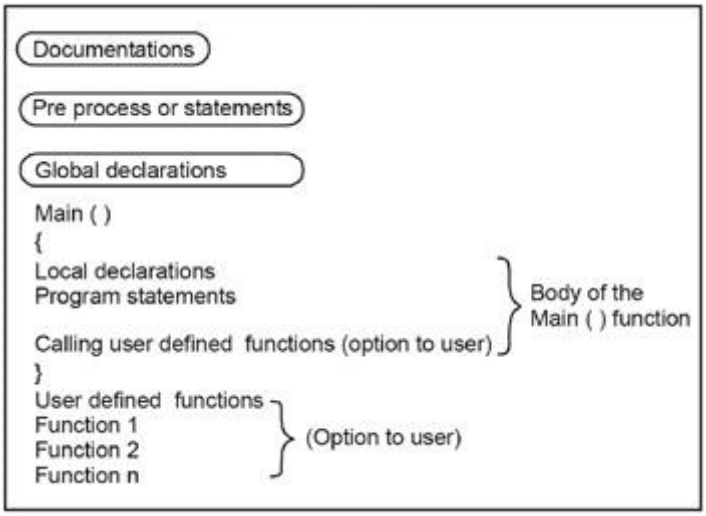
UNIT-I

Procedural	Object-oriented
procedure	method
record	object
module	class
procedure call	message

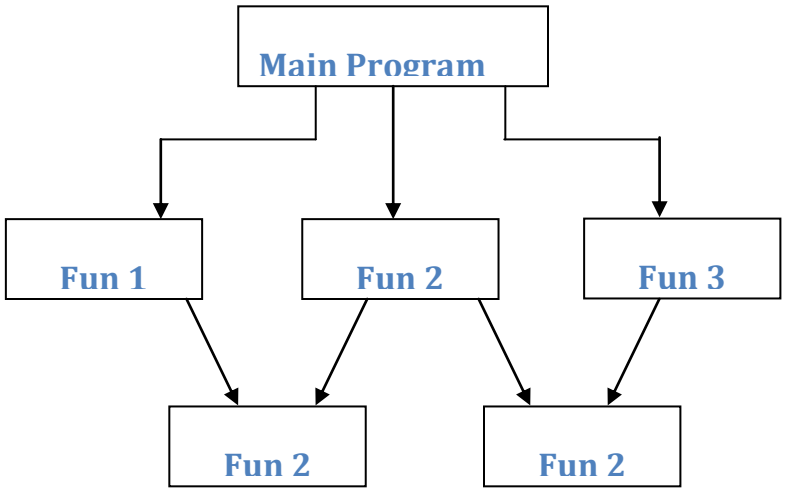


NEED FOR OBJECT ORIENTED PROGRAM:

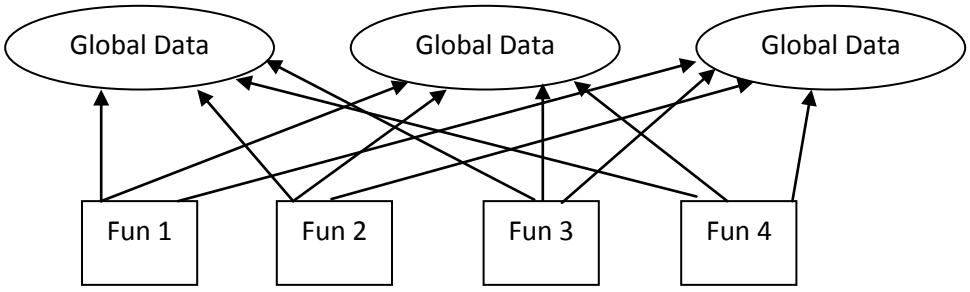
Procedure Oriented Programming (POP)

The high level languages, such as BASIC, COBOL, C, FORTRAN are commonly known as Procedure Oriented Programming.

Using this approach, the problem is viewed in sequence of things to be done, like reading, processing and displaying or printing. To carry out these tasks the function concepts must be used.



- This concept basically consists of number of statements and these statements are organized or grouped into functions.
- While developing these functions the programmer must care about the data that is being used in various functions.
- A multi-function program, the data must be declared as global, so that data can be accessed by all the functions within the program & each function can also have its own data called local data.



- Th global data can be accessed anywhere in the program. In large program it is very difficult to identify what data is accessed by which function. In this case we must revised about the external data and as well as the functions that access the global data. At this situation there is so many chances for an error.

The focus of **PROCEDURAL PROGRAMMING** is to break down a programming task into a collection of

1.variables

2. data structures

3.subroutines

Eg:

```
struct database {
  int id_number;
  int age;
  float salary;
};

int main()
{
  database employee;  //There is now an employee variable that has modifiable
                      // variables inside it.
  employee.age = 22;
```

```
  employee.id_number = 1;
  employee.salary = 12000.21;
}
```

OBJECT-ORIENTED PROGRAMMING it is to break down a programming task into objects that expose

1.behavior (methods)

2.data (members or attributes) using interfaces.

```
class exforsys
{
private:
 int x,y; //DATA
public:
void sum() //METHOD
{
…… ……
}
};
main()
{
exforsys e1;
 …… ……
 }
```

- This programming approach is developed to reduce the some of the drawbacks encountered in the Procedure Oriented Programming Approach.
- The OO Programming approach treats data as critical element and does not allow the data to freely around the program.
- It bundles the data more closely to the functions that operate on it; it also protects data from the accidental modification from outside the function.
- The object oriented programming approach divides the program into number of entities called objects and builds the data and functions that operates on data around the objects.
- The data of an object can only access by the functions associated with that object.

Difference between C & C++

S.No	Procedure oriented Programming (C)	Object Oriented Programming (C++)
1.	Programs are divided into smaller sub-programs known as functions	Programs are divides into objects & classes
2.	Here global data is shared by most of the functions	Objects are easily communicated with each other through function.
3.	It is a Top-Down Approach	It is a Bottom-Up Approach
4.	Data cannot be secured and available to all the function	Data can be secured and can be available in the class in which it is declared
5.	Here, the reusability is not possible, hence redundant code cannot be avoided.	Here, we can reuse the existing one using the Inheritance concept

Communication

BASIC CONCEPTS OF OOPS:

CLASS:

In object-oriented programming, A class is just a **template** which contains the various attributes and functions of an object of the class.

For example consider we have a Class of ***Cars*** under which

***Santro Xing, Alto* and *WaganR* represents individual Objects**.

Properties of the *Car* class

1. Model
2. Year of Manufacture
3. Colour
4. Top Speed
5. Engine Power etc.,

functions like

1. Start
2. Move
3. Stop

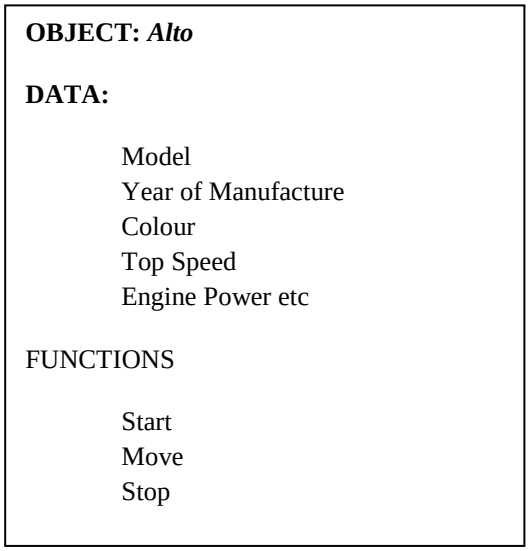
form the **Methods** of *Car* Class.

Note: No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

Classes are the most important constituents of **Object Oriented Programming**.
Another example :
consider
int a;
'a' as an object of the class 'int'.

Objects

1. An object is a run-time entity of a class.
2. Objects are identified by its unique name. An object represents a particular instance of a class.
3. There can be more than one instance of an object. Each instance of an object can hold its own **relevant data**.
4. An Object is a collection of data members and associated member functions also known as methods.



Characteristics:

1. *Object* is the basic unit of object-oriented system.
2. Objects have their own *state*.
3. Objects communicate by sending and receiving *messages*.

Encapsulation & Abstraction:

Data Encapsulation Example:

Encapsulation means combining both data and function that operates on that data into a single unit(eg classes and objects in C++(or in any othe OPP).

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example:

```
#include <iostream>
using namespace std;
class Adder{
  public:
    // constructor
    Adder(int i = 0)
    {
      total = i;
    }

    // interface to outside world
    void addNum(int number)
    {
       total += number;
    }
 // interface to outside world
    int getTotal()
    {
       return total;
    };
  private:
    // hidden data from outside world
    int total;
};
int main( )
{
  Adder a;
```

```
  a.addNum(10);
  a.addNum(20);
  a.addNum(30);

  cout << "Total " << a.getTotal() <<endl;
  return 0;
}
```

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

Total 60

The **public members addNum** and **getTotal** are the **interfaces** to the outside world and a user needs to know them to use the class.

The private member **total** is something that is **hidden** from the outside world, but is needed for the class to operate properly.

Abstraction means hiding complex details of creation(logic and functions) and just implenment/call them without bothering about how they work?.

Data Hiding means to make data invisible to external functions to minimize accidental modification/chage of important data.

```
class employee /* Encapsultaion*/
{
private:
char name[20];
int age;
float salary;

public: /*Data Hiding, Following functions access data */
void getemployee
{
cout<< "Enter name";
cin>>name;
cout<<"Enter Age";
cin>>age;
cout<< "Enter Salary";
cin>>salary;
```

```
}
void showemployee
{

cout<<"\n Name"<<name;
cout<<"\n Age"<<age;
cout<<"\n Salary"<<salary;
}
};
void main()
{
/*Abstraction:- working defined in employee class*/

employee e1;
e1.getemployee();
e1.showemployee();
}
```

Inheritance

1. Inheritance is the process of forming a new class from an existing class or *base class*.
2. The base class is also known as *parent class* or *super class.* The new class that is formed is called *derived class*.
3. Derived class is also known as a *child class* or *sub class.* Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

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

class BaseClass {
  int i;
public:
  void setInt(int n);
  int getInt();
};

class DerivedClass : public BaseClass {
  int j;
public:
  void setJ(int n);
```

```
  int mul();
};

void BaseClass::setInt(int n)
{
  i = n;
}

int BaseClass::getInt()
{
  return i;
}

void DerivedClass::setJ(int n)
{
  j = n;
}

int DerivedClass::mul()
{
  return j * getInt();
}

int main()
{
  DerivedClass ob;

  ob.setInt(10);        // load i in BaseClass
  ob.setJ(4);          // load j in DerivedClass

  cout << ob.mul();     // displays 40

  return 0;
}
```

Polymorphism:

In programming languages, *polymorphism* means that some code or operations or objects behave differently in different contexts.

For example, the + (plus) operator in C++:

```
4 + 5       <-- integer addition
3.14 + 2.0  <-- floating point addition
s1 + "bar"  <-- string concatenation!
```

In C++, that type of polymorphism is called *overloading*.

Message Passing:

Message for an object is a request for execution of procedure and therefore invoke a procedure in the receiving object that generates the desired result.

```
class A
    {
     methodA(int x)
     {

     }
    }
    class B
    {
     methodB(int y)
     {

     }
    }

class C{
main()
{
 A a;
 B b;
 a.methodA(10);
b.methodB(10);
}
}
```

a.methodA(10);

a->Object

method->message

10->information

INTRODUCTION TO C++

Basics of C++ Programming

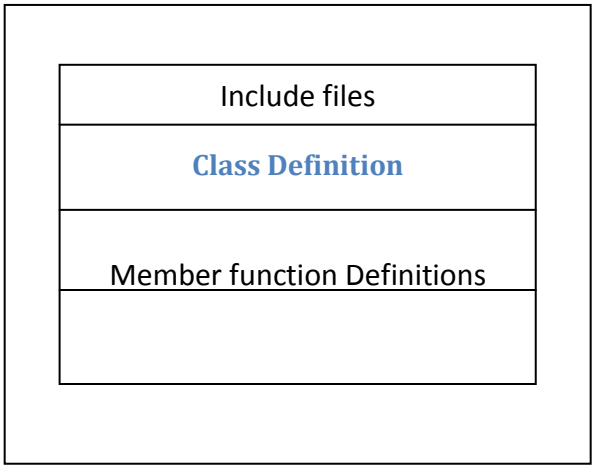
C++ was developed by BJARNE STROUSSTRUP at AT&T BELL Laboratories in Murry Hill, USA in early 1980's.

STROUSSTRUP combines the features of 'C' language and 'SIMULA67' to create more powerful language that support OOPS concepts, and that language was named as "C with CLASSES". In late 1983, the name got changed to C++.

The idea of C++ comes from 'C' language increment operator (++) means more additions.

C++ is the superset of 'C' language, most of the 'C' language features can also applied to C++, but the object oriented features (Classes, Inheritance, Polymorphism, Overloading) makes the C++ truly as Object Oriented Programming language.

Structure of C++ Program



Include files provides instructions to the compiler to link functions from the system library.

Eg: #include <iostream.h>

#include – Preprocessor Directive

iostream.h – Header File

- A class is a way to bind and its associated functions together. It is a user defined datatype. It must be declared at class declaration part.
- Member function definition describes how the class functions are implemented. This must be the next part of the C++ program.
- Finally main program part, which begins program execution.

```
main( )
{


}
```

Program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function is the logical and of the program.

Input / Output statements

Input Stream

Syntax:

```
cin >> var1 >> var2 >>;
```

cin – Keyword, it is an object, predefined in C++ to correspond to the standard input stream.

>> - is the extraction or get from operator

Extraction operation (>>) takes the value from the stream object on its left and places it in the variable on its right.

Eg:

```
cin>>x;

cin>>a>>b>>c;
```

Output Stream:

Syntax:

```
cout<<var1<<var2;
```

cout - object of standard output stream

<< - is called the insertion or put to operator

It directs the contents of the variable on its right to the object on its left.

Output stream can be used to display messages on output screen.

Eg:

```
cout<<a<<b;

cout<<"value of x is"<<x;

cout<<"Value of x is"<<x<<"less than"<<y;
```

TOKENS

The smallest individual units in a program are known as tokens.

C++ has the following tokens

- Keywords
- Identifiers
- Constants
- Strings
- Operators

Keywords

- It has a predefined meaning and cannot be changed by the user
- Keywords cannot be used as names for the program variables.

Keywords supported by C++ are:

asm	double	new	switch	
auto	else		operator	template
break	enum		private	this
case	extern		protected	throw
catch	float		public	try
char	for	register	typedef	
class	friend	return	union	
const	goto	short	unsigned	

continue	if	signed	virtual	
default	inline	sizeof	void	
delete	int	static	volatile	
do	long	struct	while	

Identifiers

Identifiers refer to the names of **variables, functions, arrays**, classes, etc. created by the programmer.

Rules for naming these identifiers:

1. Only alphabetic characters, digits and underscores are permitted.
2. The name cannot start with a digit.
3. Uppercase and lowercase letters are distinct.
4. A declared keyword cannot be used as a variable name.

(i) Variables:

It is an entity whose value can be changed during program execution and is known to the program by a name.

A variable can hold only one value at a time during program execution.

A variable is the storage location in memory that is stored by its value. A variable is identified or denoted by a variable name.

The variable name is a sequence of one or more letters, digits or underscore.

Rules for defining variable name:

 A variable name can have one or more letters or digits or underscore for example character _.

 White space, punctuation symbols or other characters are not permitted to denote variable name.

 A variable name must begin with a letter. .

 Variable names cannot be keywords or any reserved words of the C++ programming language.

- C++ is a case-sensitive language.

- Variable names written in capital letters differ from variable names with the same name but written in small letters.
- For example, the variable name EXFORSYS differs from the variable name exforsys.

Eg:

Allowable variable names	Invalid names	
i	1_B	– 1st letter must be alphabet
sum	$xy	- 1st letter must be alphabet
A_B	x+b	- special symbol '+' not allowed

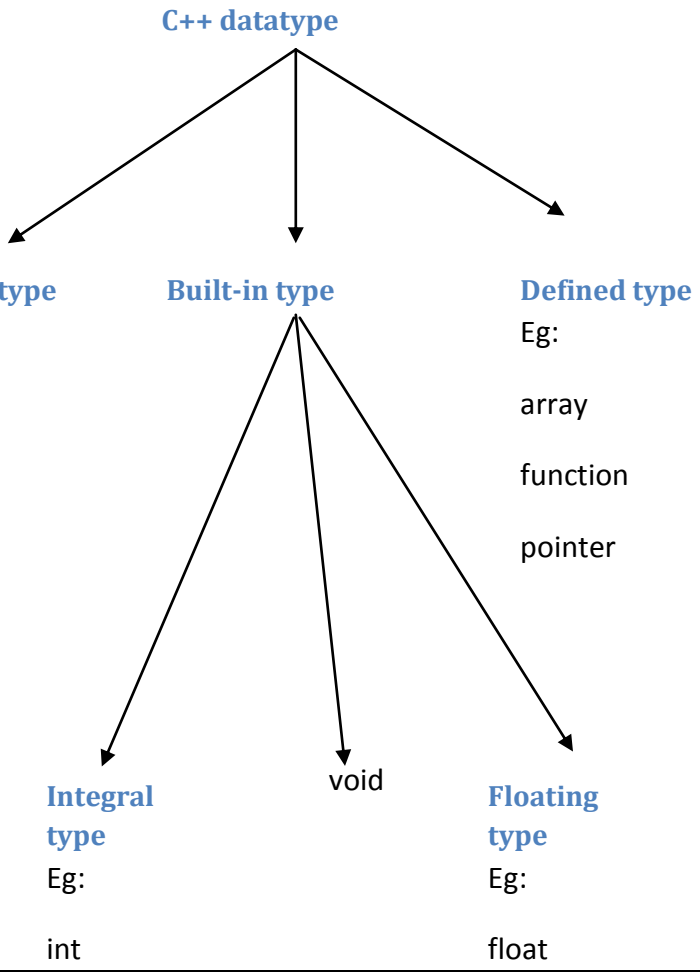
Declaration of Variables

Syntax

datatype variablename;

Datatype

It is the type of data, that is going to be processed within the program



Eg:

int x;

float y,z;

char a;

A variable can be declared anywhere in the program before its first use.

Constants

A quantity that does not change is known as constants.

- Integer constants are represented as decimal notation.
- Decimal notation is represented with a number.
- Octal notation is represented with the number preceded by a zero(0) character.
- A hexadecimal number is preceded with the characters 0x.

Types of constants:

- Integer constants - Eg: 123, 25 – without decimal point
- Character constants - Eg: ‘A’, ‘B’, ‘*’, ‘1’
- Real constants - Eg: 12.3, 2.5 - with decimal point

Strings

A sequence of characters is called string. String constants are enclosed in double quotes as follows

“Hello”

Operators

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.

Types of Operators

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increment & decrement Operators
6. Conditional Operators
7. Bitwise Operators
8. Special Operators
9. Manipulators
10. Memory allocate / delete Operators

An expression is a combination of variables, constants and operators written according to the syntax of the language.

Arithmetic Operators

C++ has both unary & binary arithmetic operators.

- **Unary operators** are those, which operate on a **single operand**.
- Whereas, **binary operators on two operands** +, -, *, /, %

Examples for Unary Operators:

1. int x = 10;

y = -x;

2. int x = 5;
   sum = -x;

Examples for Unary Operators:

1. int x = 16, y=5;
   x+y = 21; (result of the arithmetic expression)

   x-y = 11;

   x*y=80;

/ - Division Operator

Eg:

x = 10, y = 3;

x/y=3; (The result is truncated, the decimal part is discarded.)

% - Modulo Division

The result is the remainder of the integer division only applicable for integer values.

x=11, y = 2

x%y = 1

Relational Operators

- A relational operator is used to make comparison between two expressions.
- All relational operators are binary and require two operands.

<, <=, >, >=, ++, !=

Relational Expression

Expression1 (relational operator) Expression2

Expression1 & 2 – may be either constants or variables or arithmetic expression.

Eg:

a < b (Compares its left hand side operand with its right hand side operand)

10 = = 15(Equals)

a != b(Not equals)

- **An relational expression is always return either zero or 1, after evaluation.**

Eg: (a+b) <= (c+d)

arithmetic expression

Here relational operator compares the relation between arithmetic expressions.

Logical Operators

&& - Logical AND

!! - Logical OR

! - Logical NOT

Logical operators are used when we want to test more than one condition and make decisions.

Eg: (a<b) && (x= =10)

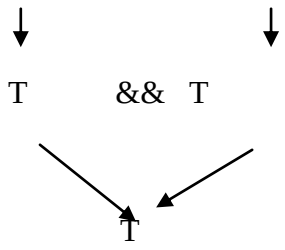
An expression of this kind, **which combines two or more relational expressions**, is termed as a logical expression.

Like simple relational expressions, a logical expression also yields a value of one or zero, according to the truth table.

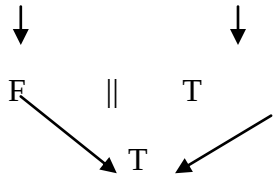
Operand 1	Operand 2	AND	OR	NOT OP1	NOT OP2
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	1
1	1	1	1	0	0

Eg:

((10 > 5) && (3 < 13))



((11 <3) || (10!= 5))



Assignment Operators

Assignment operators are used to assign the result of an expression to a variable.

Eg: a = 10;

a = a + b;

x = y;

Variable operator = operand

OP = is called as shorthand assignment operator.

V **op** = exp

is equivalent to v = v **op** exp

Eg: x+=y; ⟹x+y;

Increment & Decrement Operators

- Increment ++, this operator adds 1 to the operand
- Decrement --, this operator subtracts 1 from the operand
- Both are unary operators

Eg:

m = 5;

y = ++m; (adds 1 to m value)

x = --m; (Subtracts 1 from the value of m)

Types

- Pre Increment / Decrement OP:
- Post Increment / Decrement OP:

If the operator precedes the operand, it is called pre increment or pre decrement.

Eg: ++i, --i;

If the operator follows the operand, it is called post increment or post decrement.

Eg: i++, i--;

In the pre Increment / Decrement the operand will be altered in value before it is utilized for its purpose within the program.

Eg: x = 10;

Y = ++x;

- 1st x value is getting incremented with 1.
- Then the incremented value is assigned to y.

In the post Increment / Decrement the value of the operand will be altered after it is utilized.

Eg: y = 11;

x = y++;

- 1st x value is getting assigned to x & then the value of y is getting increased.

Conditional Operator

? :

General Form is

Conditional exp ? exp 1 : exp 2;

Conditional exp - either relational or logical expression is used.

Exp1 & exp 2 : are may be either a variable or any statement.

Eg:

(a>b)?a:b;

- Conditional expression is evaluated first.
- If the result is '1' is true, then expression1 is evaluated.
- If the result is zero, then expression2 is evaluated.

Eg: lar = (10>5)?10:5;

Bitwise Operators - Used to perform operations in bit level

Operators used:

&	-	Bitwise AND
|	-	Bitwise OR
^	-	Exclusive OR
<<	-	Left shift
>>	-	Right shift
~	-	One's complement

Special Operators

- sizeof
- comma(,)

- size of operators returns the size the variable occupied from system memory.

Eg:

```
var = sizeof(int)

cout<<var;                    Ans: 2


x = size of (float);

cout << x;                    Ans: 4

int y;

x = sizeof (y);

cout<<y;                      Ans: 2
```

Precedence of Operators

Name	Operators	Associatively
Unary Operators	-,++,--,!, sizeof	R→ L
Mul, div & mod	*, /, %	L→ R
Add, Sub	+, -	L→ R
Relational	<. <=, >, >=	L→ R
Equality	= =, !=	L→ R
Logical AND	&&	L→ R
Logical OR	||	L→ R

Example

x =10, y = 2m z = 10.5

a = (x+y) – (x/y)*z;

= 12 – 5 * 10.5;

= 12 – 52.5;

a = -42.5

Manipulators

Manipulators are operators used to format the data display. The commonly used manipulators are endl, setw.

endl manipulators

It is used in output statement causes a line feed to be inserted, it has the same effect as using the newline character "\n" in 'C' language.

```
#include <iostream.h>

main()

{
```

```
    int a=10, b=20;
    cout << "C++ language" << endl;
    cout << "A value: " << a << endl;
    cout << "B value:" << b << endl;
}
```

O/P:

C++ language

A value: 10

B value: 20

Setw Manipulator

The setw manipulator is used or specify the field width for printing, the content of the variable.

Syntax: **setw(width);**

where width specifies the field width.

```
int a = 10;
cout << " A value" << setw(5) << a << endl;
```

Output:

A value 10

Note: Remaining operators will be discussed in 4th unit.

Symbolic Constant

Symbolic constants are constants to which symbolic names are associated for the purpose of readability and ease of handling.

- #define preprocessor directive
- const keyword
- enumerated data type

#define preprocessor directive

It associates a constant value to a symbol and is visible throughout the function in which it is defined.

Syntax:

#define symbol name constant value

Eg

#define max_value 100

#define pi 3.14

The value of symbolic constant will not change throughout the program.

Const keyword

Syntax:

const datatype var = constant;

Eg: const int max = 100;

main()

{

```
        char x[max];
        ---------
        ---------
    }
```

const size = 10;

Allowable statement, default symbolic constant type is integer. Here size is of type int.

Reference Variable:

A reference variable provides an alias (alternative name) for a previously defined variable.

For example, if we make the variable sum a reference to the variable total, then sum & total can be used interchangeably to represent that variable.

A reference variable is created as follows:

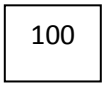
datatype & ref_name = var_name;

Eg:

```
float total = 100;
float & sum = total;
cout << sum << total;
```

Both the variables refer to the same data object in the memory i.e.

total, sum



Type Conversion:

(i) Implicit type conversion
(ii) Explicit type conversion

Implicit type conversion

It will be done by the compiler, by following the rule of lower type converted to higher type.

Eg: int y = 10;

float z = 10.5,x;

x = y+z; (y is converted to float type by compiler)

x = 10.0 + 10.5

x= 20.5 (result var. x is must be float)

Explicit type conversion

It will be performed by the programmer. According to the need of this in the program.

Syntax: datatype (var)

Eg: int y = 10;

float z = 2.5;(resultant type of y+z is float, that is converted explicitly to int type)

x = int (y + z);

Now the result is of int type.

Access specifiers:

Access specifiers are used to identify access rights for the data and member functions of the class.
There are three main types of access specifiers in C++ programming language: **private public protected**

- A*private* member within a class denotes that only members of the same class have accessibility. The *private* member is inaccessible from outside the class.
- *Public* members are accessible from outside the class.
- A protected access specifier is a stagebetween *private* and *public* access. If member functions defined in a class are *protected*, they cannot be accessed from outside the class but can be accessed from the derived class.

When defining access specifiers, the programmer must use the keywords: *private*, *public* or *protected* when needed, followed by a **colon** and then define the data and member functions under it.

```
#include<iostream.h>

#include<conio.h>

class employee /* Encapsultaion*/

{

public://Can accessible in main function

char name[20];

int age;//Variable value couldn’t be assigned

float salary;


public: /*Data Hiding, Following functions access data */

void getemployee()

{

cout<< "Enter name";

cin>>name;

cout<<"Enter Age";

cin>>age;

cout<< "Enter Salary";

cin>>salary;

}

void showemployee()

{


cout<<"\n Name"<<name;
```

```
cout<<"\n Age"<<age;
cout<<"\n Salary"<<salary;
}
};
void main()
{
/*Abstraction:- working defined in employee class*/

employee e1;
int x1;
e1.getemployee();
e1.showemployee();
cout<<e1.age;//Can access public data member if declared as public
}
```

Function and data member

- Functions can be defined either inside or outside the class.
- Defining function members outside class
    1. very simple to define functions outside class
    2. If the functions are big—define outside

Static data members of class

1. Static members are stored at a location where they are retained throughout the execution of the program and are not stored with class objects.
2. Stored only as a single copy– similar to member functions.
3. All the static data members are initialized to zero at the time of declaration.

4. One time initialization is permitted,the member cannot be reinitialized.

Ex:1

```
#include<iostream.h>
#include<conio.h>
void dummy()
{
static int i=10;
int j=20;
i+=10;
j+=5;
cout<<"i="<<i<<"j="<<j;
}
void main()
{
clrscr();
dummy();//i is initialised as 10,At the end of the call i value is 20
dummy();
getch();//Here i is not initialised to 10,instead it is 20.
}
```

Output:

```
i=20j=25
i=30j=25
```

Ex:2

```
#include<iostream.h>
#include<stdlib.h>
#include<conio.h>
class student
{

 static int PassingMark;//this does not define static member
 int SubjectMark[5];//only declation permitted
 int pass;
 public:
 void DisplayMarks()
 {
                    for (int i=0;i<5;++i)
                     {
                     cout<<"Marks of subject No.";
                     cout<<i<<"is"<<SubjectMark[i];
                     }
  }
 void SetMarks(int Marks[5])
 {
                    for (int i=0; i<5; ++i)
                       {
                       SubjectMark[i]=Marks[i];
                       }
```

```
}
 int CheckPassing()
{
          pass=1;
            for (int i=0;i<5; ++i)
               {
               if(SubjectMark[i]<PassingMark)
                     pass=0;
                  }
  if(pass)
  cout<<"congratulations! You are passing\n";
 else
 cout<<"sorry ! You are failing\n";
 return  pass;
}};
int student::PassingMark=35;
//required definition
void main()
{
//student::PassingMark=35;//Not accessible if declared private
student X;
student Y;
int Xmarks[]={75,55,65,56,89};
int Ymarks[]={15,25,100,98,89};
clrscr();
```

//X.PassingMark=35;

//Y.PassingMark=35;

X.SetMarks(Xmarks);

Y.SetMarks(Ymarks);

X.CheckPassing();

Y.CheckPassing();

}

Static members need two phases of definition

Inside the class static variable is declared as

Static int static_variable;

Outside the class it is defines as

int class_name::static_variable;

Two ways to access static data member

1.Class name::member variable name(public

2.Object. member variable name

Array of Objects:

- Array in C++ can also hold objects.
- Objects are stored in consecutive memory location.

class student

{

private:

int rollno;

string name:

```
public:
void setdetails(int roll, string stud_name)
{
rollno=roll;
name=stud_name;
}
void print()
{
cout<<”rollno”<<rollno;
cout<<”name:<<name;
}
};
void main()
{
student arrayofstudents[3];
arrayofstudents[0].setdetails(1,”xxx”);
arrayofstudents[0].print();
arrayofstudents[1].setdetails(2,”yyy”);
arrayofstudents[1].print();
arrayofstudents[2].setdetails(3,”zzz”);
arrayofstudents[2].print();
}
```

POINTER TO OBJECTS:

- C++ pointer can also have address for an objects. They ae known as pointer to objects.
- Pointer increment will increment the address by the size of the object.
- The size of the object is determined by the size of all non-static data members.

Pointer to objects use either

- star-dot combination
- arrow notation

Syntax: Class_name *Ptr_var;

Ptr_var=new student;

```
class student
{
public:
int rollno;
string name;
void print()
{
cout<<"rollno"<<rollno;
cout<<"name"<<name;
}
};
void main()
{
student *stu-ptr;
stu-ptr=new student;
(*stu-ptr).rollno=1;
```

Star-dot notation

Arrow notation

```
student *stu-ptr;
stu-ptr=new student;
stu_ptr->rollno=1;
stu_ptr->name="xxx";
```

(*stu-ptr).name=”xxx”;

(*stu-ptr).print();

}

POINTER TO MEMBERS:

Syntax:

Datatype class_name ::*ptr_to_mem=&class_name::data_member

int student :: *Rollpt=&student::rollno;

eg:

student newstudent;//object **newstudent**

student *newstudptr=&newstudent;//ptr to obj **newstudptr**

cout<<newstudent.*Rollpt;//will display 3

Pointer to member operators:

- Accessing pointer to member by an object(.*)
- Accessing pointer to member by pointer to an object(→*)

1.Accessing pointer to member by an object(.*)

int student::*Roll-pt=&student::rollno;

student newstudent;

newstudent.*Rollpt=2; //Same as newstudent.rollno=2;

cout<<newstudent.*Rollpt;//will display 2

syntax:

objectname.*pointer_to_member;

2. Accessing pointer to member by pointer to an object(→*)

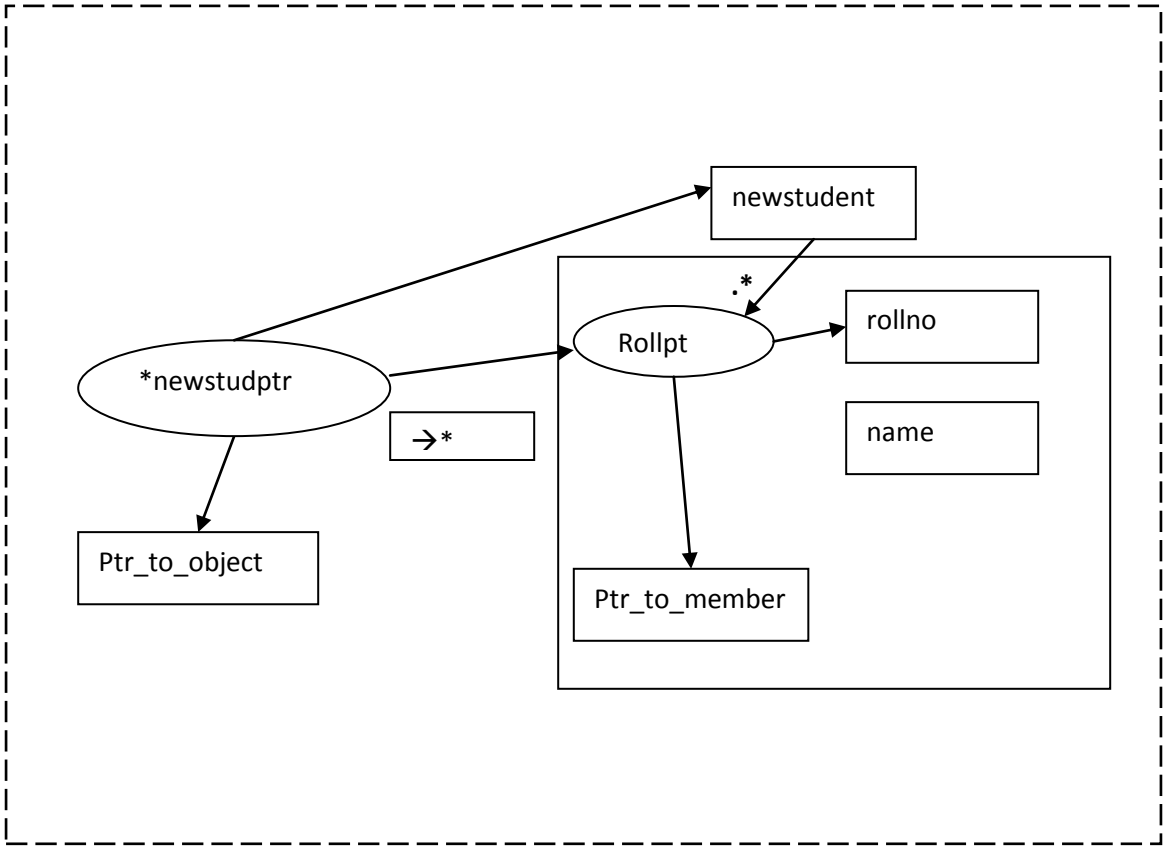
int student::*Rollpt=&student::rollno;

student newstudent;

student *newstudptr=&newstudent;

newstudptr→*Rollpt=3;

cout<<newstudptr→*Rollpt;//It Displays 3.



- Pointer to member of the class does not contain address.
- It only has the offset value when a member can be found in the class.

NESTED CLASSES:

Two classes:

Private:

- Outside class is the **nesting class** contain the entire body of the inner class.
- Inside class defined inside the outside class is known as **nested class**
- If the nested class defined as private,only nesting class can refer to it.

```
class Outside
{
private:
    class Inside
     {
    int insideInt;
    public:
        void SetInsideInt(int Tempinside)
        {
        InsideInt=TempInside;
        }
      };
public:
    int OutsideInt;
    Inside InsideObj;
    void useInside()
    {
    InsideObj.SetInsideInt(OutsideInt);
```

```
        cout<<OutsideInt;
        }
};
void main()
{
Outside Outerclass;
Outerclass.OutsideInt=5;
outerclass.useInside();
}
```

Public:

In this case it is possible to define nested class by the user.

eg:

Outside::Inside InnerPublicclass

```
//Example pgm
    class Outside
    {
    public:
        class Inside
         {
        int InsideInt;
        public:
            void SetInsideInt(int Tempinside)
```

```
            {
            InsideInt=TempInside;
            }
        };
    public:
        int OutsideInt;
        Inside InsideObj;
        void useInside()
        {
        InsideObj.SetInsideInt(OutsideInt);
        cout<<OutsideInt;
        }
};
void main()
{
Outside Outerclass;
Outerclass.OutsideInt=5;
outerclass.useInside();
Outside::Inside InnerPublicClass;
InnerPublicclass.SetInsideInt(5);
}
```

- It is also possible to define nested class outside the nesting class.

```
//Example pgm
```

```
class Outside
{
private:
class Inside;
public:
int OutsideInt;
void useInside();
};
class Outside::Inside
{
int InsideInt;
public:
                    void SetInsideInt(int Tempinside)
                    {
                    InsideInt=TempInside;
                    }
  };
void Outside::useInside()
{
                Inside InsideObj;
                InsideObj.SetInsideInt(OutsideInt);
                cout<<OutsideInt;
}
void main()
{
```

```
Outside Outerclass;
Outerclass.OutsideInt=5;
outerclass.useInside();
}
```

ASSIGNING OBJECTS:

The object assignment is of the form,

```
obj1=obj2;
//example
class student
{
  public:
  int roll;
  string name;
    void print()
    {
    }
};
void main()
{
student stud;
stud.roll=1;
stud.name=”xxx”;
stud.print();
```

```
stud stud2;

stud2=stud;  //assignment

student stud3=stud;  //Initialization

stud2.print();

stud3.print();

}
```

Difference between const variables and const object

- Constant variables are the variables whose value cannot be changed through out the program.
- But if any object is constant, value of any of the data members (const or non const) of that object cannot be changed through out the program.
- Constant object can invoke only constant function.

FUNCTIONS IN C++:

Introduction:

A class is a collection of attributes.

1.Data Attributes.

2.Function Attributes.(Member function).

- Member function can be accessed by an object by dot notation.
- <Object_name>.<Function_name>

Non-member functions:

Function which are not belong to any class is called **non-member functions**.

Non-member functions are accessed by

Function_name();

Steps involved in writing a program:

Code:

```
int Global_var;
```

```
int myfunc(int arg1,int arg2)

{

int function_var;

}

int main()

{

int main_var,value1,value2;

int result=myfunc(value1,value2) ;//Function Call

cout<<result;//Next statement after function call

}
```

Three steps involved in running the program

1. Compiling
2. Linking
3. Loading

- When a program is compiled the object code of all the functions is generated.i.e main() and myfunc() object code is generated but separately.
- When a program to be executed the object code of the program to be loaded in the memory.
- main() is automatically loaded but other functions are not loaded that time.
- When a call is made the object code of the function**(myfunc())** is loaded in the main memory.
- Then the control is transferred from **main()** to **myfunc().This process is called Linking.**
- When a function call is made ,the context of the calling function**-main() i.e** CPU register values and the values of the variables are to be stored.
- When the called function-**myfunc()** is over ,they have to be restored. This process is called **context-switching.**
- The main memory area where functions are loaded is called **stack.**

Difference between C and C++ functions

1.C++ has inline functions where the functions are small,It will eliminate the overhead of context switching.

2.The object code of C++ function contains some additional information than C functions.

Inline Functions:

Every time a function is called,it takes a lot of extra time in executing a series of instructions such as

- Jumping to function
- Saving registers
- Pushing arguments into the stack
- Returning to the calling function

To eliminate the cost of calls to small functions, C++ proposes a new feature called inline function.

An inline function is a function that is expanded inline when it is invoked.

The compiler replaces the function call with the corresponding function code .

Another alternative is to use macros**(#define).**

#define a 5

The body of the macro with required parameter replaces macro call before compilation. preprocessor does this replacement.

The drawback of macros is that usual error checking does not occur during compilation.

Example:

```
Void main()

{

Student s;

s.rollno=1;

s.name=”xxx”;

s.printdetails();

}
```

The Compiler generates the code similar to the following**(Inline function)**

```
Void main()
```

{

Student s;

s.rollno=1;

s.name=”xxx”;

//Three statements are replaced because printdetails() is treated as inline function.

Cout<<”Roll no is”<<rollno;

Cout<<”Name is”<<name;

}

Normal function:

Void main()

{

Student s;

s.rollno=1;

s.name=”xxx”;

//The address of the function printdetails() with appropriate arguments

}

It is possible to define inline functions outside of the class

Class student

{

Public:

```
Int rollno;
String name;
Void printdetails();
}
Inline void student::printdetails()
{
Cout<<”Roll no is”<<rollno;
Cout<<”Name is”<<name;
}
Void main()
{
Student s;
s.rollno=1;
s.name=”xxx”;
s.printdetails();
}
```

There are some cases when functions are not made inline

Recursive function:

A recursive function call cannot be replaced by function body.

Static variables:

C++ cannot allow inline functions with static variables

Default arguments:

C++ allows to call a function without specifying all its arguments.

- Prototype default values
    - Default values are specified when the function is declared
    - We must add default from right to left

Float amout(float principle, int period, float rate=0.15);

Value = amount(5000, 7);//function use default value of 0.15 for rate.

Value = amount(500, 5, 0.12);//It takes rate as 0.12 as explicit value

Code:

```
# include <iostream.h>
# include <conio.h>
void main()
{
        float amount;
        float value(float p, int n, float r =0.15);
        void printline(char ch='*', int len=40);
        printline();
        amount = value(5000.00, 5);
        cout << "\n " << amount << "\n";
        printline('=');
}
float value(float p, int n, float r)
{
        int year = 1;
        float sum = p;
```

```
cout<<year<<sum<<r;
}
void printline(char ch, int len)
{
    for(int i=0; i<=len;i++)
        Cout<<ch;
}
```

Functions with object as parameters:

An object can be used as a function argument. This is done in two ways.

1.A copy of the entire object is passed to the function.

This method is called **pass by value**. Since the copy of the object is passed to the function. Changes does not affect the original object.

2.Only address of the function is passed to the function.

This method is called **pass by reference.** Function works directly on the actual object.

Code:

```
class time
{
int hours;
int minutes;
public:
void gettime(int h,int m)
{
hours=h;
minutes=m;
```

```
}
void puttime()
{
cout<<”hours”<<”hours and”;
cout<<minutes<<”minutes”;
}
void sum(time,time);
};
void time:: sum(time t1,time t2)
{
minutes=t1.minutes+t2.minutes;
hours= minutes/60;
minutes= minutes%60;
hours=hours+t1.hours+t2.hours;
}
int main()
{
time t1,t2,t3;
t1.gettime(2,45);
t2.gettime(3,30);
t3.sum(t1,t2);
cout<<”t1=”;
t1.puttime();
cout<<”t2=”;
t2.puttime();
```

```
cout<<"t3=";
t3.puttime();
}
```

Function overloading:

Overloading refers to the use of same thing for different purposes.This is known as function polymorphism in c++.

- Use the same function name to create functions that perform a variety of different tasks known as function overloading
- Function overloading rules:
    - Function name should be same
    - Number of arguments should be differ otherwise arguments data type should be differ
    - Example
        1. int Add(int a, int b);
        2. int Add(int a, int b, int c);
        3. int Add(double a, double b))

code:

```
# include <iostream.h>
# include <conio.h>
int volume(int);
double volume(double, int);
long volume(long, int, int);
int main( )
{
```

```
        cout<< volume(10) << "\n";
        cout << volume(2.5, 8) << "\n";
        cout << voume(100l, 75, 15);
        return 0;
}
int volume (int s)
{
        return(s*s*s);
}

double volume(double r, int h)
{
        return(3.14*r*r*h);
}
long volume(long l, int b, int h)
{
return(l*b*h);
}
```

Friend function:

What is a Friend Function?

- A friend function is used for accessing the non-public members of a class. A class can allow non-member functions and other classes to access its own private data, by making them friends. Thus, a friend function is an ordinary function or a member of another class.
- Non- member function cannot have an access to the private data of a class

- **Income_tax()** is a common function for objects such as **manager** or **scientist**.C++ allows common functions to be made friendly with both the classes
- The function declaration should be preceded by the keyword **friend .**

How to define and use Friend Function in C++:

- The friend function is written as any other normal function, except the function declaration of these functions is preceded with the keyword friend. The friend function must have the class to which it is declared as friend passed to it in argument.

Some important points

- The keyword friend is placed only in the function declaration of the friend function and not in the function definition.

It is possible to declare a function as friend in any number of classes.

- When a class is declared as a friend, the friend class has access to the private data of the class that made this a friend.
- It is possible to declare the friend function as either private or public.
- The function can be invoked without the use of an object.

Code:

```
# include <iostream.h>

# include <conio.h>

class sample

{

        int a;

        int b;

public:

        void setvalue ()

        {

                a=20;
```

```
        b=25;
    }
    friend float mean(sample s);
}
float mean(sample s)
{
    return float(s.a +s.b)/2.0;
}
int main()
{
    sample x;
    x.setvalue();
    cout << " Mean value = " << mean(x) << "\n";
    return 0;
}
```

Static member function:

- A static function can have access to only other static members declared in the class
- A static member function can be called using the class name
    - Class name :: function name

```
Static void showcount (void)
{
    cout << "count:" << count << "\n";
}
```

Code:

```
Class test
```

```
{
        int code;
        static int count;
Public:
        Void setcode(void)
        {
                code = ++count;
        }

        Void showcode(void)
        {
                Cout << " Object number: " << code <<"\n";
        }
        Static void showcount (void)
        {
                cout << "count:" << count << "\n";
        }
};
int main()
{
        test t1,t2;
        t1.setcode();
        t2.setcode();
        test :: showcount();
        test t3;
```

```
    t3.setcode();
    test :: showcount();
    t1.showcode();
    t2.showcode();
    t3.showcode();
}
```

Following code doesn't work

```
Static void showcount (void)
    {
        cout << "count:" << code << "\n";//code is not static
    }
```

const member functions:

- A function, which guarantees not to modify the invoking object.
- If the body of the const function contains a statement that modifies the invoking object, the program does not compile.

One exception here is the mutable member. A mutable data member can be modified by const function.

```
void PrintDetails()const
   {
   cout  <<rollno;
   cout  <<name;
   cout << address;
   rollno=4    //error
    }
```

If rollno definition is changed to

mutable int rollno;

in the student class ,then there will not be an error.

The Class Constructor:

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

Following example explains the concept of constructor:

```
#include <iostream.h>

class Line
{
  public:
    void setLength( double len );
    double getLength( void );
    Line();  // This is the constructor

  private:
    double length;
};

// Member functions definitions including constructor
Line::Line(void)
{
   cout << "Object is being created" << endl;
}

void Line::setLength( double len )
{
   length = len;
}

double Line::getLength( void )
{
   return length;
}
// Main function for the program
int main( )
{
  Line line;

  // set line length
  line.setLength(6.0);
  cout << "Length of line : " << line.getLength() <<endl;
```

```
   return 0;
}
```
When the above code is compiled and executed, it produces the following result:
```
Object is being created
Length of line : 6
```
Parameterized Constructor:

A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation as shown in the following example:

```
#include <iostream.h>


class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line(double len);  // This is the constructor

   private:
      double length;
};

// Member functions definitions including constructor
Line::Line( double len)
{
    cout << "Object is being created, length = " << len << endl;
    length = len;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}
// Main function for the program
int main( )
{
   Line line(10.0);

   // get initially set length.
```

```
   cout << "Length of line : " << line.getLength() <<endl;
   // set line length again
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

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

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

Using Initialization Lists to Initialize Fields:

In case of parameterized constructor, you can use following syntax to initialize the fields:

```
Line::Line( double len): length(len)
{
    cout << "Object is being created, length = " << len << endl;
}
```

Above syntax is equal to the following syntax:

```
Line::Line( double len)
{
    cout << "Object is being created, length = " << len << endl;
    length = len;
}
```

If for a class C, you have multiple fields X, Y, Z, etc., to be initialized, then use can use same syntax and separate the fields by comma as follows:

```
C::C( double a, double b, double c): X(a), Y(b), Z(c)
{
  ....
}
```

The Class Destructor:

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Following example explains the concept of destructor:

```
#include <iostream.h>


class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line();   // This is the constructor declaration
```

```
    ~Line();  // This is the destructor: declaration

  private:
    double length;
};

// Member functions definitions including constructor
Line::Line(void)
{
   cout << "Object is being created" << endl;
}
Line::~Line(void)
{
   cout << "Object is being deleted" << endl;
}

void Line::setLength( double len )
{
   length = len;
}

double Line::getLength( void )
{
   return length;
}
// Main function for the program
int main( )
{
  Line line;

  // set line length
  line.setLength(6.0);
  cout << "Length of line : " << line.getLength() <<endl;

  return 0;
}
```

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

```
Object is being created
Length of line : 6
Object is being deleted
```

Storage Class

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. These specifies precede the type that they modify.

There are following storage classes, which can be used in a C++ Program

auto

register
static
extern
mutable

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
  int mount;
  auto int month;
}
```

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

The register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
  register int  miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C++, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

```
#include <iostream.h>

// Function declaration
void func(void);

static int count = 10; /* Global variable */

main()
{
   while(count--)
   {
     func();
   }
```

```
   return 0;
}
// Function definition
void func( void )
{
   static int i = 5; // local static variable
   i++;
   std::cout << "i is " << i ;
   std::cout << " and count is " << count << std::endl;
}
```

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

```
i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0
```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will be used in other files also, then *extern* will be used in another file to give reference of defined variable or function. Just for understanding *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.cpp

```
#include <iostream>

int count ;
extern void write_extern();

main()
{
  count = 5;
  write_extern();
}
```

Second File: support.cpp

```
#include <iostream>

extern int count;
```

```
void write_extern(void)
{
   std::cout << "Count is " << count << std::endl;
}
```

Here, *extern* keyword is being used to declare count in another file. Now compile these two files as follows:

```
$g++ main.cpp support.cpp -o write
```

This will produce **write** executable program, try to execute **write** and check the result as follows:

```
$./write
5
```

The mutable Storage Class

The **mutable** specifier applies only to class objects, which are discussed later in this tutorial. It allows a member of an object to override constness. That is, a mutable member can be modified by a const member function.

This Pointer

C++ has access to its own address through an important pointer called **this** pointer.

The**this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

Let us try the following example to understand the concept of this pointer:

```
#include <iostream.h>
class Box
{
   public:
      // Constructor definition
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
      }
      double Volume()
      {
         return length * breadth * height;
      }
      int compare(Box box)
      {
         return this->Volume() > box.Volume();
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
```

```
    double height;     // Height of a box
};

int main(void)
{
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2

  if(Box1.compare(Box2))
  {
    cout << "Box2 is smaller than Box1" <<endl;
  }
  else
  {
    cout << "Box2 is equal to or larger than Box1" <<endl;
  }
  return 0;
}
```

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

```
Constructor called.
Constructor called.
Box2 is equal to or larger than Box1
```