

Module 1

Introduction to Object Oriented Concepts

1.1 STRUCTURES

- *Structure is a programming construct in C that allows us to put together variables with different data types.*
- Structure are used in 'c' for bundling together data items that collectively describe a thing, or in some other way related to each other.
- Format for defining structure is as follows :

```
struct Tag {  
    Members  
};
```

- Example for structure definition :

```
struct date  
{  
    int d,m,y;  
};
```

- In structures variables of heterogeneous data types can be declared.
- Functions can be associated with the structure, but are defined outside the structure definition.
- Example for function associated with **date** structure.

```
void next_day(struct date *);
```

- Library programmers use structures to create new data types.
- Application programs or other library programs use these new data types by declaring variables of this data type.
- Example to declare variables of structure date.

```
struct date d1;
```

- Application programmes call the associated functions by passing structure variables or their addresses to them as follows :

```
d1.d = 31;  
d1.m = 12;  
d1.y = 2003;  
next_day(&d1);
```

Creating a New Data type Using Structures

- Creation of new data type using structures is a three-step process that is executed by the library programmer.

Step 1: Put the structure definition and the prototypes of the associated functions in a header file.

Step 2: Put the definition of the associated functions in a source code and create a library.

Step 3: Provide the header file and the library, in whatever media, to other programmers who want to use this new data type.

Using structures in Application programs

- The steps to use this new data type are as follows:

Step 1: Include the header file provided by the library programmer in the source code.

Step 2: Declare variables of new data type in the source code.

Step 3: Embed calls to the associated functions by passing these variables in the source code.

Step 4: compile the source code to get the object file.

Step 5 : link the object file with the library provided by the library programmer to get the executable or another library.

1.2 Procedure oriented programming

- In procedure oriented programming code divided into functions.
- Data is passed from one function to another to read from or written into. The focus is on procedures.

Characteristics exhibited by POP

- It focuses on process rather than data.
- It takes a problem as a sequence of things to be done such as reading, calculating and printing. Hence, a number of functions are written to solve a problem.
- A program is divided into a number of functions and each function has clearly defined purpose.
- Most of the functions share global data.
- Data moves openly around the system from function to function.

Drawbacks of POP

- It emphasis on doing things. Data is given a second class status even through data is the reason for the existence of the program.
- Data is not secure, it can be manipulated by any procedure.

- Compilers that implement the POP system do not prevent unauthorized functions from accessing/manipulating structure variables.

1.3 Object-Oriented Programming

- Lack of data security of procedure-oriented programming system led to object-oriented programming systems.
- In OOPS, we try to model real-world objects.
- Most of the real world objects have internal parts and interfaces that enable us to operate them.
- These interfaces perfectly manipulate the internal parts of the objects. They also have exclusive rights to do so.
- Object oriented programming took the best ideas of structured programming and combined them with several new concepts, hence resulted in a different way of organizing a program.
- Generally program can be organised in one of the two ways : **around its code**(what is happening) or **around its data** (who is being affected).
- Using only structured oriented programming techniques, programs are typically organised around code. This approach can be thought of as “code acting on data”.
- Object- oriented programs works the other way around. They are organized around data, with the key principle being “data controlling access to code”.
- In an object-oriented language, you define the data and the routines that are permitted to act on that data.

Features of OOPS

- It emphasis in own data rather than procedure.
- It is based on the principles of inheritance, polymorphism, encapsulation and data abstraction.
- It implements programs using the objects.
- Data and the functions are wrapped into a single unit called class so that data is hidden and is safe from accidental alternation.
- Objects communicate with each other through functions.
- New data and functions can be easily added whenever necessary.

Basic concepts of OOPS

- **Classes**
- **Objects**
- **Data abstraction and encapsulation**
- **Inheritance**
- **Polymorphism**

Objects

- Objects are the basic run time entities in an object oriented system.
- They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.
- They may also represent user-defined data such as vectors, time and lists. Program objects should be chosen such that they match closely with the real-world objects.
- Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in c.
- Objects contain data, and code to manipulate the data.

Classes

- Class is a template for an object.
- Class contain data and functions bundled together under a unit.
- The entire set of data and code of an object can be made a user-defined data type with the help of class.
- Objects are variables of the type class.
- Once a class has been defined, we can create any number of objects belonging to that class.
- Each object is associated with the data of type class with which they are created.
- A class is thus a collection of objects similar types.
- For examples, Mango, Apple and orange members of class fruit.

Data abstraction and Encapsulation

- The wrapping up of data and function into a single unit (called class) is known as *encapsulation*.
- The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.
- These functions provide the interface between the object's data and the program.
- This insulation of the data from direct access by the program is called ***data hiding or information hiding***.
- Abstraction refers to the act of representing essential features without including the background details or explanation.
- Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, wait, and cost, and function operate on these attributes.
- They encapsulate all the essential properties of the object that are to be created.

Inheritance

- Inheritance as the name suggests is the concept of inheriting or deriving properties of an existing class to get new class or classes.
- Inheriting class is called as **derived class** or child class.
- Class from which properties are derived are known as base class or parent class.
- All general features are included in the **base class**.
- Specific features are included in the child class.

- The main advantage of using inheritance in object oriented programming it helps in reducing the code size since the common characteristics is placed separately in base class and it is just referred in the derived class, hence provides **reusability** of the code.

Polymorphism

- Poly refers many. So polymorphism as the name suggests is a certain item appearing in different forms or ways.
- That is making a function or operator to act in different forms depending on the place they are present is called polymorphism.
- In OOPS functions can be made to exhibit polymorphic behaviour. Functions with different set of formal arguments can have the same name.
- Operators are overloaded to perform new operations and make them to operate on different data types.
- This is very important feature of object oriented programming methodology which extended the handling of data type and operations.

Conclusion

- The above given important features of object oriented programming among the numerous features it have gives the following advantages to the programming world.
- The advantages are :
 - Data protection or security of data is achieved by concept of data hiding.
 - Reduces program size and saves time by the concept of reusability which is achieved by the terminology of inheritance.
 - Operators can be given new functions as per user which extends the usage.

1.4 Comparison of C++ with C

C++

- C++ was developed by Bjarne Stroustrup.
- C++ is superset of C Language.
- C++ supports both POP and OOP for code development.
- C++ compiler can understand all the keywords that a C compiler can understand.
- **class** keyword is used to define a new data type.

C

- C was developed by Dennis Ritchi.
- C is subset of C++

- C supports procedural programming paradigm for code development.
- C compiler cannot understand C++ keywords.
- **struct** keyword is used to define new data type.

1.5 Console Input/Output in c++

Console Output

- The C++ stream object that is defined to access standard output is **cout**.
- '**cout**' is an object of ostream class.
- cout is used together with the **insertion operator** which is written as <<.
- The << is a left shift operator has its definition extended in c++. Here it acts as insertion operator. It is a binary operator. It takes two operands. The object on its left must be an object of the 'ostream' class. The operand on its right must be a value of some fundamental data type.
- The value on the right side of the insertion operator is inserted into the stream headed towards the device associated within the object on the left.
- It is possible to cascade 'insertion' operator.
- We can pass constants instead of variables as operands to the 'insertion' operator.
- The file '**iostream.h**' needs to be included in the source code for successful compilation since the object 'cout' and 'insertion' operator have been declared in that file.
- Example program to demonstrate '**cout**'

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

int main()
{
    int x,y;
    cout << "Enter two numbers : ";
    cin >> x >> y;
    cout << "you entered : "<< x << "\t" << y << endl;
    cout << "displaying constant value : "<< 10 << endl;
    return 0;
}
```

- **Output :**



```
Enter two numbers : 1 2
you entered : 1 2
displaying constant value : 10
```

Console input

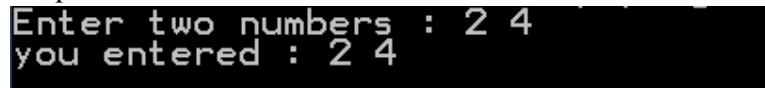
- The C++ stream object that is defined to access the standard input is **cin**.
- cin is used together with the extraction operator , which is written as >>. The operator is then followed by the variable where the extracted data is stored.
- The >> is a right shift operator whose definition is extended in C++.

- Extraction operator is a binary operator that takes two operands. Operand on its left must be some object of **istream** class, and to its right must be a variable of some fundamental data type.
- The value for the variable on the right side of the extraction operator is extracted from the stream originating from the device associated with the object on the left.
- The file '**iostream.h**' needs to be included in the source code for successful compilation since the object 'cin' and 'extraction' operator have been declared in that file.
- Example program to demonstrate '**cin**'.

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

int main()
{
    int x,y;
    cout << "Enter two numbers : ";
    cin >> x >> y;
    cout << "you entered : "<< x << "\t" << y << endl;
    return 0;
}
```

- Output



```
Enter two numbers : 2 4
you entered : 2 4
```

1.6 Variables and Reference Variables in c++

- Variables in C++ can be declared anywhere inside a function.

Reference variable

- *Reference variables are variables in their own right. They just happen to have the address of another variable.*
- *Reference variable is nothing but a reference for an existing variable.*
- It shares the memory location with an existing variable.
- Syntax for declaring a reference variable is as follows:

<data-type> & <ref-var-name> = <existing-var-name>;

- Any changes made to reference variable will reflect in the existing variable.
- Reference variable must be initialized at the time of declaration otherwise the compiler will not know what address it has to record for the reference variable).
- Example program :

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

int main()
{
    int x;
    x=10;
    cout << "value of x : "<< x <<endl;

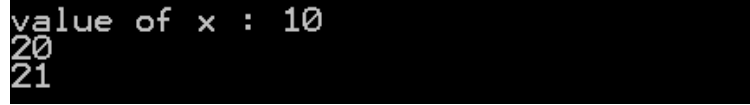
    int & iRef = x;
    iRef = 20;

    cout<<x<<endl;

    x++;

    cout<< iRef <<endl;
    return 0;
}
```

- **output**



```
value of x : 10
20
21
```

- Reference variable can be passed as function argument.

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

void increment(int &);

int main()
{
    int x;
    x=10;
    increment(x);
    cout<<"\n value of x: "<<x<<endl;
    return 0;
}

void increment(int & r)
{
    r++;
}
```

- **Output**



```
value of x: 11
```

1.7 Function Prototyping

- A prototype describes the functions interface to the compiler. It tells the compiler the return type of the function as well as the number, type, and sequence of its formal arguments.
- The general syntax of function prototype is as follows:

return_type function_name(argument_list);

- Providing names to formal arguments in function prototypes is optional. Even if such names are provided, they need not match those provided in the function definition.
- Function prototyping is necessary in C++. By making prototyping necessary, the compiler ensures the following:
 - The return value of the function is handled correctly.
 - Correct number and type of arguments are passed to a function.
- Function prototyping guarantees protection from errors arising out of incorrect function calls.
- Function prototyping produces automatic type conversions wherever appropriate.

Example program for function prototyping.

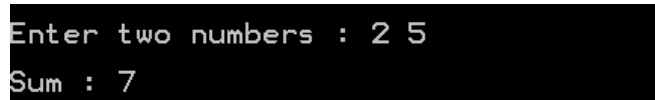
```
#include<iostream>
using namespace std;

int add(int,int);

int main()
{
    int x,y,z;
    cout << "\n\nEnter two numbers : ";
    cin >> x >> y;
    z = add(x,y);
    cout << "\n" << "Sum : " << z << endl ;
}

int add(int a,int b)
{
    return(a+b);
}
```

- output



```
Enter two numbers : 2 5
Sum : 7
```

1.8 Function Overloading

- C++ allows two or more functions to have same name.
- Function is said to be overloaded if it has same name but different signature.
- ***Signature of a function means the number, type and sequence of formal arguments of the function.***
- Depending upon the type of parameters that are passed to the function call, the compiler decides which of the available definitions will be invoked.
- Function prototypes should be provided to the compiler for matching the function calls.
- The linker, during link time links the function call with the correct function definition.
- Example program for function overloading.

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

int add(int,int);
int add(int,int,int);

int main()
{
    int x,y,z;

    x = add(10,20);
    y = add(10,20,30);
    cout << "\n" << "x : " << x << " y : " << y << endl ;
}

int add(int a,int b)
{
    return(a+b);
}

int add(int a,int b,int c)
{
    return(a+b+c);
}
```

- **Output**



```
x : 30 y : 60
```

- The compiler decides which function to be called based upon the number, type, and sequence of parameters that are passed to the function call.
- Function overloading is also known as function polymorphism because, just like polymorphism in the real world where an entity exists in more than one form, the same function name carries different meanings.
- Function polymorphism is static in nature because the function definition to be executed is selected by the compiler during compile time itself. Thus, the overloaded function is said to exhibit static polymorphism.

1.9 Introduction to Classes and Objects

classes

- As we know that in structures code(member function) and data(data member)are not bound together, hence code (member function) did not have exclusive rights to manipulate data (data member).
- To avoid the above problem structures were redefined in C++ by encapsulating both data and code, hence providing code with the exclusive right to manipulate the data members.
- Member functions are invoked in the same way as member data are accessed, that is by using the variable-to-member access operator(dot operator).
- In a member function, one can directly refer to members of the object for which the member function is invoked.
- Each structure variable contains separate copy of the data member within itself. However only one copy of the member function exists.
- Now even though code and data is encapsulated in the structure, it doesn't not mean that member function is still not given exclusive right on data member. To overcome this we will declare **data members in the structure as private** and **member functions as public**.

- Example program showing redefined structure in C++.

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

struct Distance
{
    private:
        int iFeet;
        float fInches;

    public:

    void setFeet(int x)
    {
        iFeet = x;
    }

    int getFeet()
    {
        return iFeet;
    }

    void setInches(float y)
    {
        fInches =y;
    }

    float getInches()
    {
        return fInches;
    }
};

int main()
{
    Distance d1,d2;
    d1.setFeet(2);
    d1.setInches(2.2);
    d2.setFeet(3);
    d2.setInches(3.3);
    cout<< d1.getFeet() <<" "<<d1.getInches()<<endl;
    cout<< d2.getFeet() <<" "<<d2.getInches()<<endl;

    d1.iFeet = 20; // error private data member accessed
                  // by non-member functionf

    cout<<" \n d1.iFeet : "<<d1.iFeet<<endl;
}
```

- The keyword **private** and **public** are known as ‘**access specifiers**’ or ‘**access modifiers**’ because they control the access to the members of the structure.
- The **private keyword** makes data and functions private. Private data and functions can be accessed only from inside the same class.
- The **public keyword** makes data and functions public. Public data and functions can be accessed out of the class.
- C++ introduces a new keyword **class** as a substitute for the keyword **struct**. *In a structure, member functions are public by default.* On the other hand, *class members are private by default.* This is the only difference between the **class** keyword and **struct** keyword.
- Example to redefine structure in the above example to class.

```
class Distance
{
    int iFeet;           // private by default
    float fInches;       // private by default

    public:
    void setFeet(int x)
    {
        iFeet = x;
    }

    int getFeet()
    {
        return iFeet;
    }

    void setInches(float y)
    {
        fInches =y;
    }

    float getInches()
    {
        return fInches;
    }
};
```

- A class is a blueprint for the object.
- We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.
- As, many houses can be made from the same description, we can create many objects from a class.

objects

- **Variables of classes is known as objects.**
- When class is defined, only the specification for the object is defined; no memory or storage is allocated.
- To use the data and access functions defined in the class, you need to create objects.
- Syntax :

classname objectVariableName;

Scope Resolution Operator (::)

- Scope resolution operator (::) is used to define a function outside a class or when we want to use a global variable but also has a local variable with same name.
- The scope resolution operator signifies the class to which member functions belong.
- The class name is specified on the left-hand side of the scope resolution operator.
- The name of the function is defined on the right-hand side of the operator.
- Example program for first case.

```
#include <iostream>

using namespace std;

char c = 'a';    // global variable

int main() {
    char c = 'b'; //local variable

    cout << "Local variable: " << c << "\n";
    cout << "Global variable: " << ::c << "\n"; //using scope resolution operator

    return 0;
}
```

- example program for second case.

```
class Distance
{
    int iFeet; //private by default

    public:
        void setFeet(int);
        int getFeet();
        Distance add(Distance);
};

void Distance :: setFeet(int x)
{
    iFeet = x;
}

int Distance :: getFeet()
{
    return iFeet;
}

Distance Distance :: add(Distance dd)
{
    Distance temp;
    temp.iFeet = iFeet+dd.iFeet;
    return temp;
}
```

The 'this' pointer

- Compiler call the member functions of class objects using a unique pointer known as the **'this'** pointer.
- **'this'** pointer is a constant pointer.
- It always points at the object with respect to which function was called.
- After the compiler has ascertained that no attempt has been made to access the private members of an object by non-member functions, it converts the C++ code into an ordinary c language code as follows:

1. It converts the class into structure with only data members.

Before	After
<pre>class Distance { int iFeet; //private by default public: void setFeet(int); //prototype only int getFeet(); };</pre>	<pre>struct Distance { int iFeet; };</pre>

2. It puts a declaration of the 'this' pointer as a leading formal argument in the prototypes of all member functions as follows.

Before	After
<pre>void setFeet(int); int getFeet();</pre>	<pre>void setFeet(Distance * const,int); int getFeet(Distance * const);</pre>

3. It puts the definition of the 'this' pointer as a leading formal argument in the definitions of all member functions. It also modifies all the statements to access object members by accessing them through the 'this' pointer using the pointer-to-member access operator(->).

Before	After
<pre>void Distance :: setFeet(int x) { iFeet = x; } int Distance :: getFeet() { return iFeet; }</pre>	<pre>void Distance :: setFeet(Distance * const this,int x) { this -> iFeet = x; } int Distance :: getFeet(Distance * const this) { return this -> iFeet; }</pre>

4. It passes the address of invoking object as a leading parameter to each call to the member functions.

Before	After
<pre>cout<< d1.getFeet() <<endl; cout<< d2.getFeet() <<endl;</pre>	<pre>cout<< d1.getFeet() <<endl; cout<< d3.getFeet() <<endl;</pre>

Thus every object in 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.

Data Abstraction

- Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.
- Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.
- Data abstraction is a virtue by which an object hides its internal operations from the rest of the program.
- It makes it unnecessary for the client programs to know how the data is internally arranged in the object.
- Thus, it obviates the need for the client programs to write precautionary code upon creating and while using objects.
- In C++ classes data abstraction is achieved by providing sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.
- Data abstraction is also achieved by providing access specifiers.

The Arrow Operator (->)

- Member functions can be called with respect to an object through a pointer pointing at the object using arrow operator (->).
- The dot operator is applied to the actual object. The arrow operator is used with a pointer to an object.
- The definition of arrow operator has been extended in C++. It takes not only data members on its right as in C, but also member functions as its right-hand side operand.
- If the right hand side operand is member function of a class where a pointer of the same class type is its left hand side operand, then the compiler simply passes the value of the pointer as an implicit leading parameter to the function call.

- Example

```
int main()
{
    Distance d1;
    Distance * dptr;
    dptr = &d1;

    dptr -> setFeet(1); // calling member function
                      //converts to setFeet(dptr,1);

    cout << dptr -> getFeet()<<endl; // converts to getFeet(dptr);
}
```

- In the above example '**this**' pointer also points at the same object at which the **dptr** points.

1.10 Member Functions and Member Data

Overloaded Member Function

- Member functions can be overloaded just like non-member functions.
- Function overloading enables us to have two or more functions of the same name and same signature in two different classes or in same class.
- Example for function overloading in same class.

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

class A
{
    public :
        void show();
        void show(int);
};

void A::show()
{
    cout<<"Hello\n";
}

void A::show(int x)
{
    for(int i=0;i<x;i++)
        cout<<"Hello\n";
}

int main()
{
    A A1;
    A1.show();
    A1.show(3);
    return 0;
}
```

- Example program to show function overloading in two different classes.

```
class A
{
    public :
        void show();
};

class B
{
    public :
        void show();
};

void A::show()
{
    cout<<"class A show() function\n";
}

void B::show()
{
    cout<<"class B show() function\n";
}

int main()
{
    A A1;
    B B1;
    A1.show();
    B1.show();
    return 0;
}
```


- A function of the same name show() is defined in both the classes – ‘A’ and ‘B’. The signature is also same.
- But compiler will not be confused because of ‘this’ pointer.
- The function prototype in the respective classes will be actually as follows:

```
void show(A * const);  
void show(B * const);
```

Default values for Formal Arguments of Member Functions

- As default arguments can be assigned to non-member functions, we can also assign default arguments for member functions.
- Example program :

```
#include<iostream>  
using namespace std;  
  
class A  
{  
public :  
    void show(int=1);  
};  
void A::show(int x)  
{  
    for(int i=0;i<x;i++)  
        cout<<"Hello\n";  
}  
  
int main()  
{  
    A A1;  
    A1.show();  
    A1.show(3);  
    return 0;  
}
```

- first call to show() function will take default value, whereas in second call to show() function default value is overridden.
- Output will be as follows:



```
Hello  
Hello  
Hello  
Hello
```

- While overloading member functions care should be taken while specifying default values for some or all of its default arguments.
- Example demonstrating ambiguity :

```
class A  
{  
public :  
    void show()  
    void show(int=0); // ambiguity error  
};
```

- Default values must be specified in the function prototype and not in function definitions.
- Only the last argument must be given default value. You cannot have a default argument followed by non-default argument.
- If you default an argument, then you will have to default all the subsequent arguments after that.
- You can give any value a default value to argument, compatible with its data type.

Inline Member Functions

Inline Functions

- Inline functions are used to speed of execution of the executable files.
- C++ inserts calls to the normal functions and the inline functions in different ways in an executable.
- ***Inline function is a function whose compiled code is 'in line' with the rest of the program.***
- compiler replaces the function call with the corresponding function code.
- With inline code, the program does not have to jump to another location to execute the code and then jump back.
- Hence, inline functions run faster than regular functions.
- For specifying an inline function, you must
 - Prefix the definition of the function with the inline keyword and
 - Define the function before all functions that call it.
- Example program

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

inline int max(int a,int b)
{
    return a>b?a:b;
}

int main()
{
    cout<< max(10,20);
    cout<< " "<<"\n"<<max(99,88)<<"\n";
    return 0;
}
```
- Compiler will expand inline function only if it is not :
 - Recursive
 - Having any looping constructs
 - Having static variables in the function.
- If the function satisfies the above condition then the compiler will expand the inline function else it will call such functions in the ordinary fashion.

Inline Member Functions

- Member functions are made inline by either of the following methods.
 - By defining the function within the class itself.
 - By only prototyping and not defining the function within the class. The function is defined outside the class by using the ***scope resolution operator***. The definition is prefixed by the ***inline*** keyword.

- Example program :

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

class A
{
    public:
        void show();
}

inline void A:: show()
{
    // definition of show()
}
```

Constant Member Functions

- Constant member functions cannot change the value of the data members.
- Member functions are specified as constants by suffixing the prototype and the function definition header with the const keyword.
- For constant member functions, the memory occupied by invoking object is a read only memory.
- For constant member functions, the 'this' pointer becomes 'a constant pointer to a constant'
- Constant member functions can be called with respect to constant object or non-constant object.
- Example program :

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

class A
{
    int x;
    int y;

    public :
        A()
        {
            x=1;
            y=1;
        }

        void abc() const;
        void def();
};
```

```

void A :: abc() const
{
    x++; // error !!
    cout<<"abc() : value of x: "<<x<<endl;
    y++; // error !!
    cout<<"abc() : value of y: "<<y<<endl;
}

void A :: def()
{
    x++;
    cout<<"def() : value of x: "<<x<<endl;
    y++;
    cout<<"def() : value of y: "<<y<<endl;
}

int main()
{
    A A1;
    A1.abc();
    A1.def();
    return 0;
}

```

Mutable Data Member

- Mutable data member is never constant.
- It can be modified inside constant member functions also.
- Prefixing the declaration with the keyword mutable makes it mutable.
- Example

```

class A
{
    int x;
    mutable int y;

public :
    A()
    {
        x=1;
        y=1;
    }

    void abc() const;
    void def();
};

void A :: abc() const
{
    x++; //error
    cout<<"abc() : value of x: "<<x<<endl;
    y++; // ok : can modify mutable data member in a constant member function
    cout<<"abc() : value of y: "<<y<<endl;
}

void A :: def()
{
    x++;
    cout<<"def() : value of x: "<<x<<endl;
    y++;
    cout<<"def() : value of y: "<<y<<endl;
}

```

Friends

- A class can have global non-member functions and member functions of other classes as friends.
- Such functions can directly access the private data members of objects of the class.

Friend Non-member functions

- *A friend function is a non-member function that has special rights to access private data members of any object of the class of whom it is a friend.*
- A friend function is prototyped within the definition of the class of which it is intended to be a friend.
- The prototype is **prefixed** with the keyword *friend*.
- Non-member friend functions are defined **without using** the scope resolution operator.
- Non-member friend functions are **not called** with respect to an object.
- Important points about friend function are :
 - *friend* keyword should appear in the prototype only not in the definition.
 - Friend function can be prototyped in either the private or the public section of the class.
 - Friend function takes one extra parameter as compared to a member function that performs the same task. This is because it cannot be called with respect to any object. Instead, object itself appears as an explicit parameter in the function call.
 - We should not use scope resolution operator while defining a friend function.
- There are situations where a function that needs to access the private data members of the objects of a class cannot be called with respect to an object of the class. In such situations the function must be declared as a friend.

- Example program :

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

class A
{
    int x;
    public :
        A()
        {
            x=1;
        }
        friend void abc(A&);
};

void abc(A& Aobj)
{
    Aobj.x++;
    cout<<"x value modified by friend non member function : "<<Aobj.x<<"\n";
}

int main()
{
    A A1;
    abc(A1);
}
```

Friend Classes

- A class can be a friend of another class.
- *Member functions of a friend class can access private data members of objects of the class of which it is a friend.*
- If class B is to be made friend of class A, then the statement

friend class B;

should be written within the definition of class A.

- The above statement can be placed in private or public section of the class.
- The member functions of friend class can access private data members of objects of class to which it is a friend.
- Important property of friendships is that they are not transitive: The friend of a friend is not considered to be a friend unless explicitly specified.

- Example program :

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

class A
{
    int a;
public:
    A()
    {
        a=0;
    }
    friend class B;    // Friend class
};

class B
{
    int b;
    A *aptr;
public:
    void showA(A& x)
    {
        // Since B is friend of A, it can access
        // private members of A
        std::cout << "\n A::a = " << x.a<<endl;
    }
};

int main()
{
    A a;
    B b;
    b.showA(a);
    return 0;
}
```

Friend Member Function

- We can make specific member functions of class as friend to another class.
- Example program

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

class A; // forward declaration to avoid circular dependence

class B
{
    A * Aptr;
public:
    void test_friend(int=0);
};

class A
{
    int x;
public:
    friend void B::test_friend(int=0);
};

inline void B::test_friend(int p)
{
    Aptr->x = p;
    cout<<"\n x value = "<<Aptr->x<<endl;
}

int main()
{
    A Aobj;
    B Bobj;
    Bobj.test_friend(22);
}
```

Friends as Bridges

- Friend functions can be used as bridges between two classes
- Suppose there are two unrelated classes whose private data members need a simultaneous update through a common function. This functions should be declared as a friend to both the classes.
- Example :

```
class B;
class A
{
    friend void ab(const A&, const B&);
};
class B
{
    friend void ab(const A&, const B&);
};
```

Static Members

Static Member Data

- Static data members hold global data that is common to all objects of the class.
- Ex of such global data are
- Count of objects currently present,
- Common data accessed by all objects, etc.
- Static data members are **members of the class and not of any object of the class**, that is, they are not contained inside any class.

- To declare a static variable, prefix the declaration of the variable with **static** keyword and make it static member of the class.

- Ex

```
class A
{
    static float x;
};
```

- Memory for static members are not allocated inside class, neither when objects are declared. This is because a static data member is not a member of any object.
- We need to explicitly define static data member outside the class to allocate memory.
- EX:
float A :: x;

- The above statement initializes x value to zero.
- If some other initial value is desired instead, the statement should be written as follows:

```
float A :: x = 1.1;
```

- Making static data members **private** prevents any change from non-member functions as only member functions can change the value of static data members.
- Static data members does not increase the size of objects of the class. Static data members are not contained within objects.
- **There is only one copy of the static data member in the memory.**
- Static data members of integral type can be initialized within the class itself if the need arises.
- Static data members that has been initialized inside the class must be still defined outside the class to allocate memory for it.
- Once the initial value has been supplied within the class, the static data member must not be re-initialized when it is defined.
- Member functions can refer to static data members directly.
- The dot operator can be used to refer to the static data member of the class with respect to an object.
- The class name with the scope resolution operator can do this directly.

- Example program

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

class A
{
    static int count;
    int x;

    public:
    void setcounter()
    {
        count++;
        x=count+10;
        cout<<"\ncount : "<<count<<endl;
        cout<<"\nx      : "<<x<<endl;
    }
};

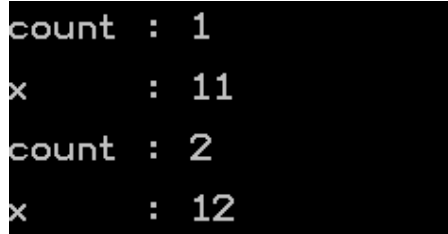
int A :: count = 0;
```



```
int A :: count = 0;

int main()
{
    A a1,a2;
    a1.setcounter();
    a2.setcounter();
    return 0;
}
```

- Output



```
count : 1
x      : 11
count : 2
x      : 12
```

- Few things that static data members can do but non-static data members cannot
 - A static data members can be of the *same type* as the class of which it is a member.
 - A static data member can appear as *default value* for the formal arguments of member functions of its class.
 - A static data member can be declared to be a constant. In that case, the member functions will be able to only read it but not modify its value.

Static Member Functions

- Static member functions are defined to modify static data members of the class.
- Prefixing the function prototype with **keyword static** specifies it as a static member function.
- The keyword static *should not appear* in the definition of the function.
- Static member functions can be called directly without an object.
- Static member functions do not take the 'this' pointer as a formal argument.
- Therefore accessing non static members through a static member function results in compile time errors.
- *Static member functions can access only static data members of the class.*
- Static member functions can still be called with respect to objects also.
- Example program.

```

#include<iostream>
using namespace std;
class st
{
    int code;
    static int count;

public:
    st()
    {
        code=++count;
    }

    void showcode()
    {
        cout<<"\n\tobject number is : "<<code<<endl;
    }

    static void showcount()
    {
        cout<<"\n\tcount objects : "<<count<<endl;
        cout<<"\n\tobject number is : "<<code<<endl;// error
    }
};
int st::count;
int main()
{
    st obj1,obj2;

    obj1.showcount();
    obj1.showcode();
    obj2.showcount();
    obj2.showcode();
    return 0;
}

```

1.10 Namespaces

- Namespace is a container for identifiers. It puts the names of its members in a distinct space so that they don't conflict with the names in other namespaces or global namespace.
- Namespaces enable the c++ programmer to prevent pollution of the global namespace that leads to name clashes.
- Syntax :

```

namespace MySpace
{
    // Declarations
}

int main() {}

```

- Example program

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

// first name space
namespace first_space{
    void func(){
        cout << "Inside first_space" << endl;
    }
}

// second name space
namespace second_space{
    void func(){
        cout << "Inside second_space" << endl;
    }
}

int main () {

    // Calls function from first name space.
    first_space::func();

    // Calls function from second name space.
    second_space::func();

    return 0;
}
```

Rules to create namespace

- The namespace definition must be done at the global scope, or nested inside another namespace.
- Namespace definition doesn't terminate with a semicolon like in class definition.
- You can use alias name for a namespace name.

```
namespace StudyTonightDotCom
{
    void study();
    class Learn { };
}

namespace St = StudyTonightDotCom;    // St is now alias for StudyTonightDotCom
```

- You cannot create instance of namespace.
- A namespace definition can be continued and extended over multiple files, they are not redefined or overridden.

```
Header1.h

namespace MySpace
{
    int x;
    void f();
}

Header2.h

#include "Header1.h";
namespace MySpace
{
    int y;
    void g();
}
```

- There are three ways to use namespace in program.
 - The scope resolution operator.
 - The using directive.
 - The using declaration.

The Scope resolution operator

- Any name declared in a namespace can be explicitly specified using the namespace's name and the scope resolution operator with the identifier.

```
namespace MySpace
{
    class A
    {
        static int i;
        public:
        void f();
    };
    class B;        // class name declaration
    void func();    // global function declaration
}
```

```
int MySpace::A::i=9;    // Initializing static class

class MySpace::B
{
    int x;
    public:
    int getdata()
    {
        cout << x;
    }
    B();    // Constructor declaration
}

MySpace::B::B()    // Constructor definition
{
    x=0;
}
```

The using directive

- Using keyword allows you to import an entire namespace into your program with a global scope. It can be used to import namespace into another namespace or any program.

```
Namespace1.h

namespace X
{
    int x;
    class Check
    {
        int i;
    };
}

Namespace2.h

include "Namespace1.h";
namespace Y
{
    using namespace X;
    Check obj;
    int y;
}
```

```
Program.cpp

#include "Namespace2.h";
void test()
{
    using Namespace Y;
    Check obj2;
}
```

1.11 Nested Classes

- A class can be defined inside another class. Such a class is known as a **nested class**.
- The class that contains the nested class is known as the **enclosing class**.
- A nested class is a member and as such has the same access rights as any other member.
- Nested classes can be defined in the private, public and protected portions of the enclosing class.
- The size of objects of an enclosing class is not affected by the presence of nested classes.
- Member functions of nested class can be defined outside the definition of the enclosing class.
- A nested class may be only prototyped within its enclosing class and defined later.
- Objects of the nested class are defined outside the member functions of the enclosing class, by using the name of the enclosing class followed by the scope resolution operator.
- An object of the nested class can be used in any of the member functions of the enclosing class without the scope resolution operator.

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

class A
{
    class B
    {
        public:
            void BTest( );
    };
    B B1;

    public :
        void ATest();
};

void A :: Atest()
{
    B1.Btest();
    B B2;
    B2.BTest();
}
```

- An object of the nested class can be a member of the enclosing class.
- Only the public members of the nested class objects can be accessed unless the enclosing class is a friend of the nested class.
- Member functions of the nested class can access the non-static public members of the enclosing class through an object, a pointer, or a reference only.

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

class A
{
    class B
    {
        public:
            void BTest(A&);
            void BTest1();
    };
    public :
        void ATest();
};

void A :: B :: BTest(A& Aref)
{
    Aref.Atest();  // OK
}

void A :: B :: BTest1()
{
    ATest();  //Error
}
```

4.1 Constructors

A **constructor** is a kind of member function that initializes an instance of its class. A **constructor** has the same name as the class and no return value. A **constructor** can have any number of parameters and a class may have any number of overloaded **constructors**. A constructors gets called automatically for each object that has just got created. The constructor is a non static member function. It is called for an object. It therefore takes 'this' pointer as a leading formal argument.

The prototype of constructor is

<class name> (<parameter list>);

For example we want the value of data member 'finches' of each object of the class 'Distance' to be between 0.0 to 12.0 all the times within the life time of the object then introducing a suitable constructor enforces this condition.

The compiler embeds a call to the constructor for each object when it is created.

For example

```
/*Program 4.1*/
/* beginning of the class Distance.h */
Class Distance
{
```

```
Int iFeet;  
Float inches;  
Public:  
Distance(); //constructor  
Int getFeet()  
{return iFeet;}
```

```
Float fInches()  
{return fInches;}
```

```
/* rest of the class distance */  
};  
//Beginning of Distance.cpp  
#include "Distance.h"  
Distance::Distance()  
{  
iFeet=0;  
fInches=0.0;  
}  
/* Definitions of rest of the functions of class distance */  
/*end of Distance.cpp */
```

```
//Beginning of DistTest.cpp  
#include<iostream.h>  
#include "Distance.h"  
Void main()  
{  
Distance d1;  
Cout<<d1.getFeet()<<" "<<d1.getInches();  
} //End of Distance.cpp  
Output 0 0 0.
```

In the statement
Distance d1;
Constructor called implicitly by the compiler.

The Zero argument constructor

The constructor which does not take any argument is called as **zero-argument constructor or default constructor**.

In the above program

```
Distance(); //is Zero argument constructor  
Distance d1; // Calls the constructor automatically
```

As an another Example

```
//Program 4.2  
// Beginning of A.h
```



```
Class A
{
Int x;
Public:
A();
Void setx(const int -1);
Int getx();
};
//End of A.h

//Beginning of A.cpp
#include "A.h"
#include<iostream.h>
A::A()
{cout<<"Constructor of A is called";}

//Beginning of AMain.cpp
#include "A.h"
Void main()
{
A a1;
Cout<<"End of Program\n";}
//End of AMain.cpp
```

Output
Constructor A is called

Parameterized Constructors

If constructor has arguments then it is called as parameterized constructor. The parameterized constructors can take argument and can be overloaded. For example

```
//beginning of Distance.h
//Program 4.3
Class Distance
{
Public:
Distance(); //Prototype for default constructor
Distance(int, float); //Prototype for parametrized constructor
Int getFeet();
Float fInches();
//Rest of the class distance
};

//Beginning of Distance.cpp
#include "Distance.h"
Distance :: Distance()
```

```
{
iFeet =0;
fInched=0.0;
}
Distance::Distance(int p, float q) // Definition of parameterized constructor
{
iFeet =p;
fInched=q;
}

//Beginning of DistTest.cpp
#include<iostream.h>
#include "Distance.h"
Void main()
{
Distance d1(2,1.5); // parameterized constructor called
Cout<<d1.getFeet()<<" "<<d1.getInches();
} //End of Distance.cpp
Output 2 1.5
```

In The above program 4.3 the constructor is called by creating an object in the stack. Below program creates the object in the heap.

```
//Beginning of distTest2.cpp
//Program 4.4
#include<iostream.h>
#include "Distance.h"
Void main()
{
Distance *dPtr;
dPtr=new Distance(4,6.2); // parameterized constructor called
Cout<<d1.getFeet()<<" "<<d1.getInches();
} //End of Distance.cpp
Output 4 6.2
```

Copy constructor

The copy constructor is a special type of constructor which copies one object to another object. The copy constructor is called for object being created.

For e.g.

A a1;

A a2=a1; //copy constructor is called

Or

A A2(A1); Copy constructor called

Or

A = Aptr = new A (A1); Copy constructor is called

The prototype and definition of the default copy constructor defined by the compiler as follows.

Class A

{

Public:

A(A&); // the default copy constructor

};

A::A(A& Aobj) // the default copy constructor

{ *this=Aobj;} Copies passed object into invoking object

The statement

A A2=A1;

Is converted as

A A2;

A2.A(A1); Copy constructor is called for A2 and A1 is passed as parameter to it.

4.2 Destructors

Destructors deinitializes the member data and frees up the resources acquired by the object during its lifetime.

~<class name>();

A1 ~A1(); //Destructor is called. But not legal c++ code

Explicit call to the destructor is not allowed. The above statement is transformed by the compiler as ~A(&A1).

Destructor is called for an object which is dynamically created as follows.

A=APtr;

APtr = new A;

....

....

Delete APtr;

This statement deletes the object and destructor is called.

For example the destructor is defined in the following way

Class Distance

{

Public:

Distance(); //Prototype for default constructor

~Distance(); //Prototype for destructor

//Rest of the class distance

};

//Beginning of Distance.cpp

#include "Distance.h"

```
Distance ::Distance()
{Cout<<" Constructor is called";}

Distance :: ~Distance()
{Cout<<" Destructor is called";}
//Beginning of DistTest.cpp
#include<iostream.h>
#include "Distance.cpp"
Void main()
{
Distance d1; // parameterized constructor called
} //End of Distance.cpp
```

Output :
Constructor is called
Destructor is called