Classes and objects

Limitation of structure

A class is an extension of the idea of structure in C. It is a new way of creating and implementing a user defined data type. We know that structures provide a method for packing together data and of different types. A structure is a convenient tool for handling a group of logically related data items. Once structure has been defined, we can create variables of that type using declaration that are similar to built in data type declarations.

```
struct student
{
        char name[25];
        int age;
        float weight;
};
        struct student s;       // C style
                or
        student s;              //C++ style
```

The standard C does not allow the struct data type to be treated like built-in data types. For example

```
        struct complex
        {
                float real, imag;
        };
        struct complex c1,c2,c3;
```

The complex numbers c1,c2,c3 can easily be assigned values using the dot operator. But we cannot add two complex numbers or subtract one from the other. That is c3=c1 + c2; is illegal in C.

Also data hiding is not permitted in C, that is structure members can be directly accessed by the structure variable by any function anywhere in their scope.

C++ supports all the features of structures as defined in C. But C++ has expanded its capabilities further to suit its OOP philosophy. It attempts to bring the user defined types as close as possible to the built in data types, and also provides a facility to hide the data. In C++, a structure can have both variables and functions as members. It can also declare some of its members as private so that cannot be accessed directly by the external functions.

Encapsulation with structures

In C++, there may be two types of members.

1. Data member and
2. Member functions.

Functions defined within structure can operate any member of structures. The following simple program shows the use of structures with member data and function both.

```
#include<iostream.h>
#include<conio.h >
 struct  coordinate
{
      int x;     //x coordinate
      int y;     //y coordinate
      void read()
	{
		cout<<"x coordinate";   cin>>x;
		cout<<"y coordinate";  cin>>y;
	}
	void print()
	{
		cout<<"("<<x<<","<<y<<")";

	}

	void add(coordinate p1, coordinate p2)
	{
		x=p1.x+p2.x;
		y=p1.x+p2.y;
	}
};  //end of structure definition
void main()
{
	coordinate p1,p2,p3;
	cout<<"enter co-ordinates of 1st point"<<endl;
	p1.read();
	cout<<"enter co-ordinate of points p2"<<endl;
	p2.read();
	p1.print();
	p2.print();
	p3.add(p1,p2);//p3=p1+p2
	p3.print();
	getch();
}
```

- In main(),p1.read();executes the member function read() defined in the structure co-ordinate
- The data values for p1 are assigned with input values.
- The statement p1.print( ) ; displays data member with message passed as function argument.
- p3.add(p1,p2) adds the contents of corresponding data members of p1 and p2 and assign the sum to p3
- Structure members are public by default
- The structures are extended in C++ and the new type class is defined for OOP.

C++ incorporates all these extension in another user defined type called as class. There is very little syntactical difference between structure and classes in C++, so they can be used interchangeably with minor modifications. However, in most cases class is used for holding data and function in C++. The only difference between a structure and a class in C++ is that, by default, the member of a class is private while, by default, the members of a structure are public.

Components of Class

A class is user-defined data type that includes different member data and associated member functions to access on those data. In OOP, data and function are put into a class for data hiding. When defining a class, we are creating a new abstract data type that can be treated like a built in data type. Generally a class has two parts:

i) Class declaration

ii) Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented. The general form of class declaration is

```
    class classname
    {
          private:
               datatype variable;
               -------------- ;
         public:
          return_type function_name(arguments…)
          {
               function body;
          }
    };         //end of class
```

The class declaration is similar to a struct declaration. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These function and variables are collectively called as members. They are grouped under two sections, private and public. By using the keyword "private" we can hide data which can be accessed by functions defined within the class. The keyword "public" is used so that public members can be accessed by any function even outside the class. The keyword private and public are known as visibility labels. By default, the members of a class are private. **In OOP, generally data are made private and functions are made public**.

The variables declared inside the class are known as data members and the functions are known as member functions. Only the member function can have access to the private data members and private functions. However, the public members can be accessed outside the class.

An object is an instance of the class just like as variable of a structure. The class describes the attributes of the object.

The member function needs to be defined within the class, just as an inline functions. *Function definition within class is automatically considered to be the inline so there is no need of associated keyword inline inside class for function definition.*

- Space in memory is allocated at the time when object are defined for class specification and not at the time of class specification.
- Member functions are accessible only through the object of that class using the class member access operator dot (.).
- The object of a class are defined as

```
classsname objectname;
```

A sample program

```
//an example of object
#include<iostream.h>
#include<conio.h>
class sampleobj
{
     private:
          int data;
```

```
	public:
		void getdata(int d)
		{ data=d;}

	void showdata()
	 { cout<<"Data is" <<data<<endl;}
};

void main()
{
	sampleobj s1,s2; //instance of class, i.e. object
	s1.getdata(1024);
	s2.getdata(2024);

	s1.showdata();
	s2.showdata();
	getch();
}
```

- the function getdata(int d) has one argument. When object accesses this member function as s1.getdata(1024); value 1024 is passed to function and it is assigned to the member data of calling object i.e. s1.
- The function showdata() has no argument so it displays the content of the value member data of corresponding object tht accesses it
  i.e. s1.showdata(); //prints 1024 here

```
//string as class member
#include<iostream.h>
#include<conio.h>
#include<string.h>

class student
{
	private:
		char name[25];
		int age;
		float weight;
	public:
		void setdata(char sname[],int sage, int sweight)
		{
			strcpy(name,sname);
			age=sage;
			weight=sweight;
		}

		void showdata()
		{
			cout<<"\nName="<<name;
			cout<<"\nAge="<<age;
			cout<<"\nWeight="<<weight;
		}
};

void main()
{
      student s1,s2;
      s1.setdata("Ram Bdr",18,65.5);
      s2.setdata("Sita Devi",28,55.09);

      s1.showdata();
      cout<<endl;
      s2.showdata();
      getch();
}
```

Arrays of Objects

As we know an array is a collection of similar data types. We can have an array of user defined data types class. Such variables are called arrays of objects. Like a structure we can use array of objects of a class.e.g.

```
 class student
 {
       int  roll;
       char name[20];
       public:
            void getdata(); //member function
            void putdata();
};
```

the array for objects are declared as student s[20];//array of 20 students ie array of objects.
To access member data by objects we use array index as s[i].getdata();s[i].putdata(); where i may be 0 to n-1 in array of size n..

```
//an example of object
#include<iostream.h>
#include<conio.h>
class arrayobj
{
      private:
            int data;
   public:
      void getdata(void)
      {
            cout<<"\nEnter data";
            cin>>data;
      }

      void showdata()
      { cout<<"Data is" <<data<<endl;}
};

void main()
{
      arrayobj s[5]; //array of objects
      for(int i=0;i<5;i++)
```

```
            s[i].getdata();
      for(i=0;i<5;i++)
            s[i].showdata();
      getch();
}
```

ACCESSING CLASS MEMBER : use dot operator as

Objectname.Datamember

Name of object dot operatotor(Member access specifier)

Accessing Member Functions

Objectname.functiionname(actual arguments);

Example:

```
#include<iostream.h>
#include<string.h>
class student
{
    private:
          int roll_no;
          char name[20];
          char phone[20];
    public:
          void getdata();
          {
                cout <<"enter roll no:";cin>>roll_no;
                cout<<"enter name";cin>>name;
                cot<<"enter phone no";cin>>phone;
          }

          void showdata()
          {
                cout<<"roll no";<<roll_no;
                cout<<"Name:"<<name;
                cout<<"Phone no:"<<phone;
          }
};  //end class

void main()
{
    student s1,s2,s3;
    cout<<enter records for student1";
    s1.getdata();
    cout<<"enter the record for s2";
    s2.getdata();
    s3.getdata();
    //display
    s1.showdata();
    s2.showdata();
    S3.showdata();
}
```

Defining member function:

Member functions can be defined in two places

- outside the class definition
- inside the class definition

1. **Inside the class definition**: The member function defines inside the class are considered as an inline automatically and no need of keyword inline. The example of member functions showdata(), getdata() etc in above examples are defined inside the class definition.
2. **Outside the class definition:** The member functions that are declared inside the class can be defined outside the class. The function definition outside a class definition consists of the function header with associated class label to represent the membership of that class and contains the body of the function.

The general syntax of the function definition outside class definition will be as:

```
return-type classname::function name(arguments..)
  {
        function Body;
  }
```

The :: operator tells the compiler that reticular function is member of that class.:: is called scope resolution operator.

An example

```
//an example of function
//definition outside the class
#include<iostream.h>
#include<conio.h>
class student
{
   private:
            int roll;
            char name[20];
            char phone[10];
      public:
            void getdata();//function declaration
            void showdata();

};   // end of class

// definition of functions outside class
void student::getdata()
 {
      cout<<"\nEnter Roll No:";
      cin>>roll;
      cout<<"\nEnter Name:";
      cin>>name;
      cout<<"\nEnter Phone:";
      cin>>phone;
}
void student::showdata()
 {
            cout<<"name"<<name<<endl;
            cout<<"roll no"<<roll<<endl;
            cout<<"phone"<<phone<<endl;
}
void main()
{
      student s1,s2;
      s1.getdata();
      s2.getdata();
      cout<<"first student"<<endl;
      s1.showdata();
      cout<<"second student"<<endl;
      s2.showdata();
      getch();
}
```

Nesting of member of functions

The member functions can call another member function within its definition which is called nesting of member functions

Example:

```
//nesting member function
#include<iostream.h>
#include<conio.h>
class set
{
      private:
            int m,n;
      public:
            void input();
            int largest();
            void display();
};
inline int set::largest()
{
      if(m>=n)
            return m;
      else
            return n;
}

inline void set::input()
{
      cout<<"input values of m & n"<<endl;
      cin>>m>>n;
}
void set::display()
{
      cout<<"largest value="<<largest()<<endl;
}
void main()
{
      set set1;
      set1.input();
      set1.display();
}
```

Private Member Function and Its Access

```
//accessing private member function
#include<conio.h>
#include<iostream.h>
class sample
{
      private:
            int m;
            void read()
            {
            cout<<"m="<<endl;
                  cin>>m;
            }
      public:
            void update();
            void write()
            {
            cout<<m;
      }
};

void sample:: update()
{
      read();//without dot operator
}

void main()
{
      sample s1;
      s1.write();  //legal
      //if we write s1.read();// it will be illegal
      s1.update(); //legal
}
```

The member functions have some special characteristics that are often used in program development.

- Several different classes can use the same function name. The ‘membership label ‘ will resolve their scope.
- Member functions can access the private data of the class. A non member function cannot do so. Exception-**friend function**
- a member function can call another member function directly without using the dot operator(as above read() inside update)
- the private member functions and data cannot be accessed by the object of the class directly

- A function definition inside class definition as default behaves as inline but if it is defined outside class definition, we should use keyword **inline** to make it inline function.

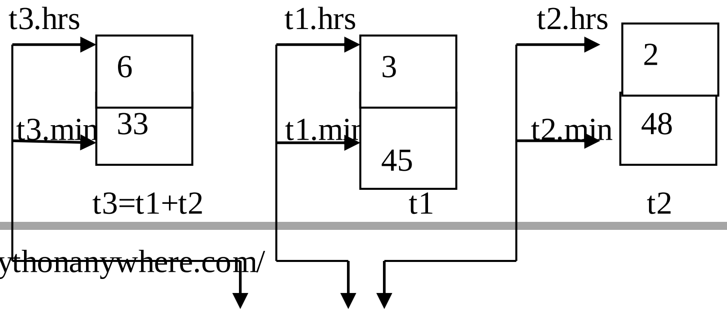
Object as function arguments:

The object can be passed as function arguments. This can be done in three ways.

- A copy of entire object is passed to the function (pass by value)
- The object name is passed as reference (pass by reference)
- Only address of the object can be transferred to the function (pass by address)

Example: (Pass by value)

```
//object as function argument
//pass by value
#include<iostream.h>
#include<conio.h>
class time
{
      int hrs,min;  //by default private.
      public:
            void gettime(int h, int m)
            { hrs=h; min=m;}
            void puttime()
            {
                  cout<<hrs<<":"<<min;
            }
            void sum(time,time);//declaration
};// end class time

void time::sum(time t1,time t2)
{
      min=t1.min+t2.min;
      hrs=min/60;
      min=min%60;
      hrs+=t1.hrs+t2.hrs;
}
void main()
{
      time t1,t2,t3;
      t1.gettime(3,45);
      t2.gettime(2,48);
      t3.sum(t1,t2);
      t1.puttime();cout<<" + ";
      t2.puttime();cout<<" = ";
      t3.puttime();
      getch();
}
```



t3.sum(t1,t2)

member accessed inside the function sum()

Passing objects by reference: Passing objects as reference is same as the passing variable function arguments. Just pass the reference of an object defining reference object as arguments.

The following program illustrates this idea

```
//object as function argument
//pass by reference

#include<iostream.h>
#include<conio.h>
class account
{
      private:
            int accno;
            float balance;
      public:
            void getdata()
            {
                  cout<<"enter account no";
                  cin>>accno;
                  cout<<"enter balnce";
                  cin>>balance;
            }
            void display()
            {
                  cout<<"account number is "<<accno<<endl;
                  cout<<"Balance is"<<balance<<endl;
            }
            void MoneyTransfer(account & acc,float amt);
            //function for transfer amount to some object passed
}; //end class
            void account::MoneyTransfer(account &acc, float amt)
            {
                  balance=balance-amt;//deduct money from balance
                  acc.balance=acc.balance+amt;//add money to destination
            }

void main()
{
      int money;
      account acc1,acc2;
      acc1.getdata();
      acc2.getdata();
      cout<<"A/C info: "<<endl;
      acc1.display();
      acc2.display();
      cout<<"how much money is to be transferred from acc2 to acc1";
      cin>>money;
      acc2.MoneyTransfer(acc1,money);//transfers money
               //from acc2 to acc1;
      cout<<"updated information of account: "<<endl;
      acc1.display();
```

```
	acc2.display();
	getch();
}
```

Passing objects by pointer: The object s can be passed as function arguments using pointer like passing normal pointer variable. The member of the class are accessed by using->operator if objects are passed using pointers. When the objects are passed using pointer, the address of the objects are passed rather than whole object. Modifying the members of the class within class affects the modification of actual data members.

For passing objects using pointer, the declaration and definition of the function `MoneyTransfer( account &acc, folat amt)` in above program can be modified slightly as

```
//Declaration
void MoneyTransfer(Account *acc, float amt);
//defn
void account::Moneytransfer(Account *acc,float amt)
{
	balance=balance-amt;
	acc->balance=acc->balance +amt;
}
```

this function can be accessed by an object as

```
acc2.MoneyTransfer(&acc1,money);
```

```
//object as function argument
//pass by address

#include<iostream.h>
#include<conio.h>
class account
{
	private:
		int accno;
		float balance;
	public:
		void getdata()
		{
			cout<<"enter account no";
			cin>>accno;
			cout<<"enter balnce";
			cin>>balance;
		}
		void display()
		{
			cout<<"account number is "<<accno<<endl;
			cout<<"Balance is"<<balance<<endl;
		}
		void MoneyTransfer(account *acc,float amt);
}; //end class
```

```
            void account::MoneyTransfer(account *acc,float amt)

            {
                  balance=balance-amt;//deduct money from balance
                  acc->balance=acc->balance + amt;//add money to destination
            }

void main()
{
      int money;
      account acc1,acc2;
      acc1.getdata();
      acc2.getdata();
      cout<<"A/C info: "<<endl;
      acc1.display();
      acc2.display();
      cout<<"how much money is to be transferred from acc2 to acc1";
      cin>>money;
      acc2.MoneyTransfer(&acc1,money);//transfers money
               //from acc2 to acc1;
      cout<<"updated information of account: "<<endl;
      acc1.display();
      acc2.display();
      getch();
}
```

Returning Object from Function:

A function can return an object. Modifying the above program slightly we can achieve this as

```
// return object by function
#include<iostream.h>
#include<conio.h>
class time
{
      int hrs,min;
      public:
            void gettime(int h,int m)
            {hrs=h;min=m;}
            void puttime()
            {cout<<hrs<<":"<<min;}
            time sum(time t2);//return object of time
};
//definition  of sum.
time time::sum(time t2)
{
      time total;
      total.min=min+t2.min;
      total.hrs=total.min/60;
      total.min=total.min%60;
      total.hrs+=hrs+t2.hrs;
      return total;  //returns object total;
}
void main()
{
      time T1,T2,T3;
      T1.gettime(2,56);
```

```
	T2.gettime(3,45);
	T3=T1.sum(T2);//passing object  T2 and returns total to T3
	cout<<"now the time value is"<<endl;
	T1.puttime(); cout<<"+";
	T2.puttime(); cout<<"=";
	T3.puttime();
	getch();

}
```

Friend function:

The concept of encapsulation and data hiding dictate that non-member functions should not be allowed to access an object's private and protected members. This policy is, if you are not a member you cannot get it. Sometimes this feature leads to considerable inconvenience in programming. If we want to use a function to operate on objects of two different classes, then a function outside a class should be allowed to access and manipulate the private members of the class. In C++, this is achieved by using the concept of friend function.
Private member of a class cannot be accessed from outside the class. Non member function of a class cannot access the member of a class. But using friend function we can achieve this.

The function declaration must be prefixed by the keyword friend whereas the function definition must not. The function could be defined anywhere in the program similar to any normal C++ function. Function definition does not use either the keyword friend or scope resolution operator **::**
A friend function is not a member of any classes but has the full access to the member of class within which it is declared as friend.

```
Class test
 {
	......
	.......
	.......
  Public:
	..........
	..........
	friend void friendfunc();//declaration
};
```

A friend function has the following characteristics.

- It is not in the scope of the class within which it has been declared as friend.
- Since it is not in the scope of the class, it cannot be called using the object of that class. It can be invoked like a normal C++ function without the help of any object.
- Unlike member functions, it cannot access member name directly and has to use an object name and dot operator with each member name. i.e. t1.x.
- It can be declared as public or private part of the class, the meaning is same.
- Normally, it takes objects as arguments.

Friends as bridges

If we want to operate on objects of two different classes, the function may take objects of two classes as arguments, and operate on their private data. For this we have to make use of friend functions that can act as bridge between two classes.

```
// Bridging classes with friend fuctions
#include<iostream.h>
#include<conio.h>
class second; //declaration like function prototype
class first
{
      private:
            int data1;
      public:
            void setdata(int x)
            {     data1=x;}
            friend int sum(first a, second b);//friend function
};
class second
{
      private:
            int data2;
   public:
      void setdata(int x)
      {     data2=x;}
      friend int sum(first a, second b);//friend function
};

int sum(first a, second b)
{
      return (a.data1 + b.data2);
}

void main()
{
      first a;
      second b;
      a.setdata(15);
      b.setdata(10);
      cout<<"sum of first and second is:"<<sum(a, b);//displays 25
      getch();
}
```

Example

```
// an example of friend function
#include<iostream.h>
#include<conio.h>
class time
{
      private:
            int hrs,min;
      public:
            void gettime(int h,int m)
            {hrs=h;min=m;}
            void puttime()
            {cout<<hrs<<":"<<min;}
            friend time sum(time, time);//return object of time

};
//definition  of sum.
time sum(time t1, time t2)
{
      time t3;
      t3.min=t1.min+t2.min;
      t3.hrs=t3.min/60;
      t3.min=t3.min%60;
      t3.hrs+=t1.hrs+t2.hrs;
      return t3;  //returns object t3;
}
void main()
{
      time T1,T2,T3;
      T1.gettime(2,56);
      T2.gettime(3,45);
      T3=sum(T1, T2);//friend function
      cout<<"now the time value is"<<endl;
      T1.puttime(); cout<<"+";
      T2.puttime(); cout<<"=";
      T3.puttime();
      getch();
}
```

Friend Classes

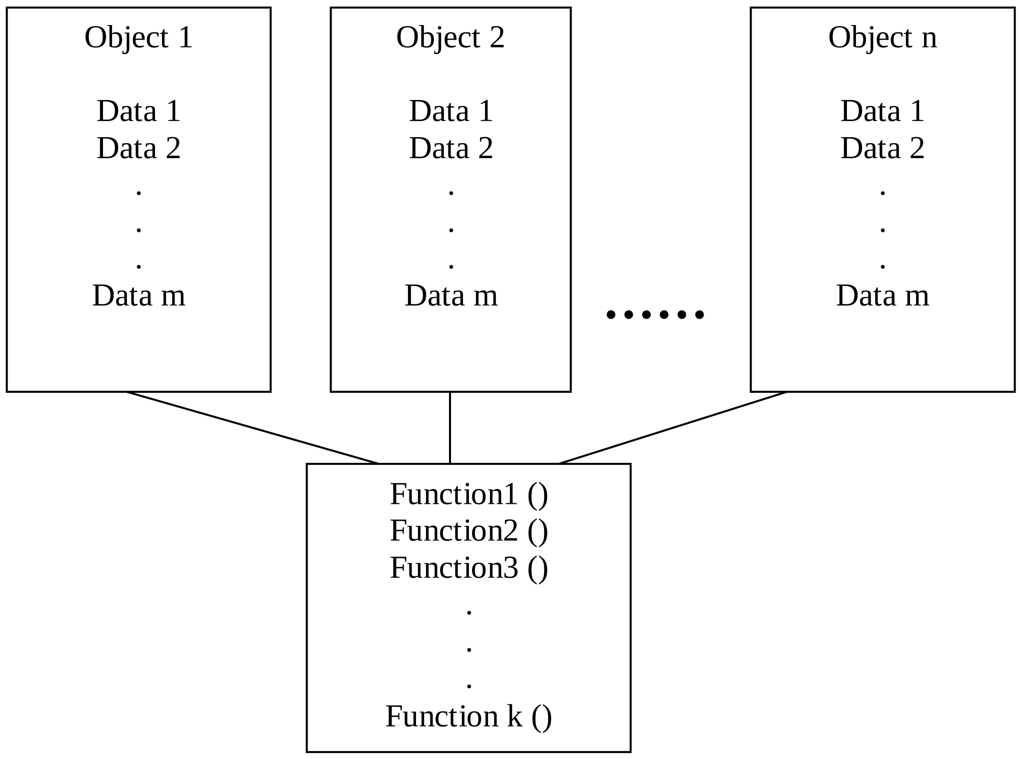
The member function of a class can all be made friends at the same time when we make the entire class a friend.

```
// an example of friend class
#include<iostream.h>
#include<conio.h>

class first
{
      private:
            int data1;
      public:
            void setdata(int x)
            {     data1=x;}
            friend class second; //class second can access private data
};
class second
{
   public:
      void func1(first a)
      {cout<<"\n data1="<<a.data1;}

      void func2(first a)
      {cout<<"\n data1="<<a.data1;}
};

void main()
{
      first a;
      second b;
      a.setdata(15);
      b.func1(a);
      b.func2(a);
      getch();
}
```

Classes, Objects, and Memory

When a class is declared, memory is not allocated to the data member of the class. So there exists a template, but data members cannot be manipulated unless an instance of this class is created by defining an object. When an object is created, memory is allocated only to its data members but not to member functions.

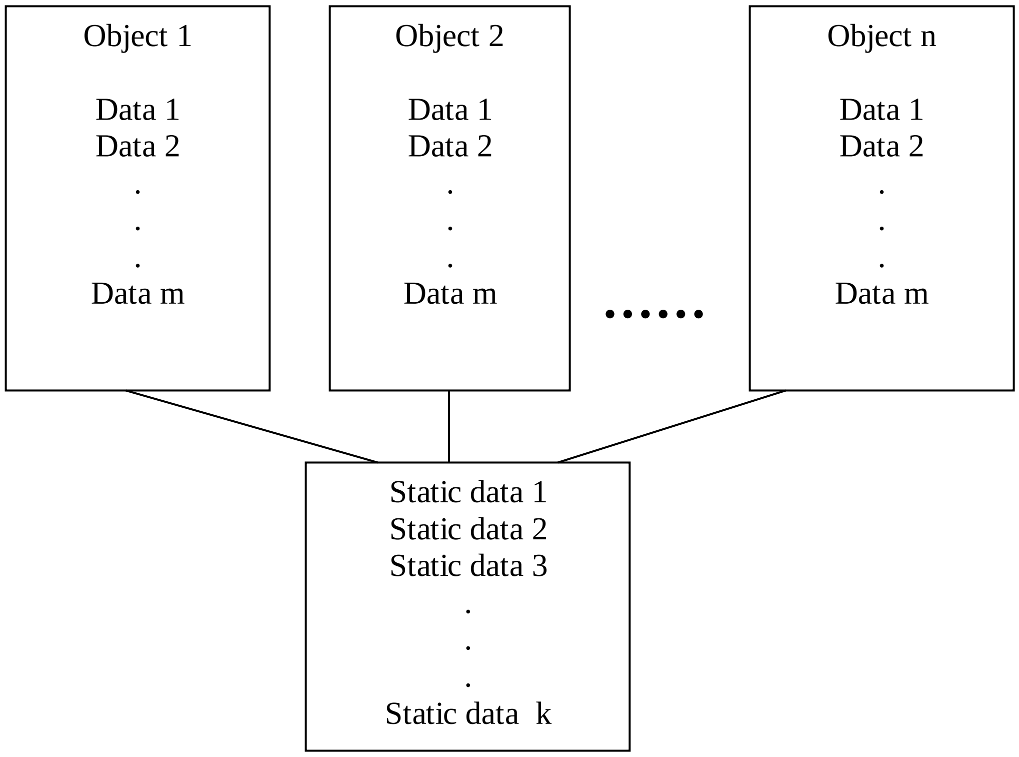
Member functions are created and stored in memory only once when a class specification is declared. All objects of that class have access to the same area in the memory where the member functions are stored. It is logically true as the member functions are same for all objects and there is no point in allocating a separate copy for each and every objects created using the same for all objects and there is no point in allocating a separate copy for each and every object created using the same class specification. However, separate storage is allocated for every object's data members since they contain different values.



It can be observed that 'n' objects of the same class are created and data members of those objects are stored in distinct memory locations, whereas the member functions of object 1 to object n are stored in the same memory area. Therefore, each object has a separate copy of data members and the different objects share the member functions among them.

Static Data as class member:

When a class is instantiated, memory is allocated to the created object. But there exist an exception to this rule. Storage space for the data members which are declared as static is allocated by once during the class declaration. All the objects of this class have access to this data member, that is all instances of the class (objects) access the same data member. When one of them modifies the static data member, the effect is visible to all the objects.



In all ‘n’ objects of the same class, automatic data members of each objects are stored in distinct memory location, where as static data members of all objects are stored in the same memory location . Thus each object has a separate copy of the automatic data members and they share static data members among them.
The type and scope of each static member variable must be defined outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object. Since they are associated with the class itself rather than with any class object, they are also known as class variables.
datatype classnme::datamember=initialization; // initialization optional

Like general static variable, static member data re defined by keyword static. The syntax for defining datamember in a class is

```
  class Classname
  {     ......
      static datatype  datamember;//declaration
     ............
 };
```

datatype classnme::datamember=initialization;

```
//an example of static data
#include<iostream.h>
#include<conio.h>

class static_data
{
      private:
            static int count;
   public:
      static_data() {count++;} //increments when object created
      int getcount() {return count;}
};
int static_data::count=0;
void main()
{
      static_data s1,s2,s3;
      cout<<"count is"<<s1.getcount()<<endl; //count is 3 i.e. static data
      cout<<"count is"<<s2.getcount()<<endl;
      cout<<"count is"<<s3.getcount()<<endl;
      cout<<"count is"<<s1.getcount()<<endl;
      getch();
}
```

Private Static Data member: A static data item is useful when all objects of same class must share common information. It has visibility within class but lifetime is the entire program.

Example:

```
//static data members
#include<iostream.h>
#include<conio.h>
class info
{
      private :
            static int count;
            int number;
      public :
            void setval(int num)
            {
                  number=num;
                  ++count;
            }

            void show()
            { cout<<"call of setval() made "<<count<<" times"<<endl;
            }
};//count is shared by all objects.

int info::count=0;//defn and initialization, linker allocates storage for it
void main()
{
      info obj1,obj2,obj3;
      obj1.show();
      obj1.setval(100);
      obj1.show();
      obj2.setval(200);
      obj3.setval(300);
      obj3.show();
      obj3.setval(500);
      obj2.show();
      getch();
}
```

```
//output
call of setval() made 0 times
call of setval() made 1 times
call of setval() made 3 times
call of setval() made 4 times
```

Hence, count is shared by all objects and updated by setval, when setval() is called by any object of that class. Whatever the data member, private, public or protected, it must always be defined using scope resolution operator**::**. Such variable acts as bridge between several objects of same class.

Access Rule for static data member:
If data member are public static, then they can be accessed by using :: operator or member access operator through objects as

```
class info
{
      public:
          static public_int;
      private:
          static private_int;
};
void main()
{
      info::public_int=120;//ok
      info::private_int=120;//illegal
      info obj;
      obj.public_int=20;//ok
      obj.private_int=20;//illegal
}
```

Static Function: A static function is one which is declared as static in a class. A static function can access only static member data and can be accessed by using class name rather than object name.
Following example shows the static function as class member

```
//static function
```

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

class staticfun
{
      private:
            static int count; // count objects.
            int id;
      public:
            staticfun ()            // constructor
            {
                  count++;
                  id=count;
            }

            ~staticfun ()           // destructor
            {
                  count--;
                  cout<<"Destroying ID number"<<id<<endl;
            }
            static void show()      // static function
            {
                  cout<<"No of object is:"<<count<<endl;
            }
            void showid()
            {
                  cout<<"ID number is:"<<id<<endl;
            }
};
int staticfun::count=0; // defn of count.

void main()
{
      staticfun s1;
      staticfun:: show();
      staticfun s2,s3;
      staticfun:: show();
      //---------------
      s1.showid();
      s2.showid();
      s3.showid();
      getch();
      cout<<"-------END------"<<endl;
}
```

• When a data member is declared as static, there is only one such value for the entire class. All objects of the class share the same data.

• To access such static data, we use static function that need not refer by any object and can be called by class name with scope resolution operator(::) as

    class name:: static function();

The output of above program will be now:

```
No of object is: 1
No of object is: 3
ID number is: 1
ID number is: 2
ID number is: 3
 ——— END ———
Destroying ID number3
Destroying ID number2
Destroying ID number1
```

CONSTRUCTORS AND DESTRUCTORS

A class encapsulates both data and functions manipulating them into a single unit. It can be further used as an abstract data type for defining a class instance known as object. As like standard data types, there must be a provision of initializing objects of a class during their definition itself. A class in C++ may contain two special member functions dealing with the internal working of a class. These functions are the constructors and the destructors. A constructor enables an object to initialize itself during creation and the destructor destroys the object when it is no longer required, by releasing all the resources allocated to it.

Definition:

A constructor is a special member function of a class whose task is to allocate the required memory as well as initialize the objects of its class. A constructor has the same name as its class. For example,

```
class sample
{
     private:
          --------------
          --------------
     public:
          sample(); //constructor
};

sample ::sample()    //note no return type required
{
    //Body of constructor if defined outside the class
}
```

A constructor has no return value specification. The constructor is such member function which is executed automatically when an object is created for that class.

When an object is created, the following process will take place.

- The object occupies space at a particular time. Instantiation of an object always involves reserving enough memory space for the data of that object.
- The instantiation does not reserve the memory for the methods. They exist only once for class, not once for every object.
- In addition to the reservation of space, the constructor may also be extended to other processes for initialization of data of that object.
- Constructor is a function (member) which is called automatically when object is created.

The C++ run time system makes sure that the constructor of a class is the first member function to be executed automatically when an object of the class is created. In other words, the constructor is executed every time an object of that class is defined. It is of course possible to define a class which has no constructor at all, in such a case, the run-time system calls a dummy constructor that performs no action when its object is created.

```
//constructor
//automatic initialization is carried out using a special
//member function.
#include<iostream.h>
#include<conio.h>



class counter
{
      private:
            int count;
      public:
            counter() {count=0;}
                  //constructor initialize value of count 0
                  //same name as class name
            void inc_count(){count++;}
            int get_count(){return count;}
};

void main()
{
      counter c1,c2;  //automatically count=0
      cout<<"\n c1="<<c1.get_count();
      cout<<"\n c2="<<c2.get_count();
      c1.inc_count();
      c2.inc_count();
      c2.inc_count();
      cout<<"\n c1="<<c1.get_count();      //c1=1
      cout<<"\n c2="<<c2.get_count();      //c2=2
      getch();
}
```

In this sample example, there is a constructor whose task is to create objects and initialize them by value count=0. The constructor counter(); is automatically invoked when object c1 and c2 of class measure are created and both objects are initialized to given value as defined by constructor.

```
//implementation of stack
#include<iostream.h>
#include<stdlib.h>
#include<conio.h>
#define max 20

class stack
{
      private:
            int s[max];
            int top;
      public:
            stack()   //constructor
            { top=-1;}

       void push(int x)//put number on stack
       {

                  s[++top]=x;
       }
       int pop()//take number from stack
       {

            return s[top--];
       }
};

void main()
{
      stack s;//top of stack is -1 when object s is created
      s.push(11);
      s.push(22);
      s.push(33);
      cout<<"\nNumber Popped"<<s.pop(); //33
      cout<<"\nNumber Popped"<<s.pop(); //22
      s.push(44);
      cout<<"\nNumber Popped"<<s.pop(); //44
      getch();
}
```

Types of constructor

There are three types of constructors:

1. The default constructor
2. User-defined constructor
3. Copy constructor

1. **Default constructor:** This constructor is called implicit constructor.
   - The compiler provides a (hidden) default constructor that has no arguments.
   - The default constructor takes no arguments and performs no processing other than reservation of memory.
   - This constructor is always called by compiler if no user-defined constructor is provided.
   - This constructor is automatically called while creating the object.

Example:
```
class Account
{
     private:
     ---------
     public:
     --------
};
Account acc1, acc2;  //default constructor Account() //(hidden) is
automatically called.
```

2. **User-defined constructor:**

If initialization of data of objects is required while creating an object, then the programmer has to define his own constructor for that purpose. The name for constructor should always be same as class name. The code of a user-defined constructor does not actually cause memory to be reserved for the data because it is still done by default constructor automatically. So code is concern only for initialization of data.
It is possible to have more than one constructor in a class providing that different versions are defined by different in the argument list. This gives more flexibility in the way when object of a class are instantiated. This is known as constructor overloading.

3. **Copy constructor**:
- ➢ The copy constructor creates an object as an exact copy of another object in terms of its attributes.
- ➢ The parameter of a constructor can be of any of the data types except an object of its own class as value parameter. However, a class's own object can be passed as a reference parameter. The constructor having a reference parameter is known as copy constructor.
- ➢ In copy constructor, one newly instantiated object equals another existing object of the same class.
- ➢ It uses the = (assignment) operator when creating new object. Hence newly created object inherit the attributes of existing object.

```
class Test
{
     private:
          ---------
          ---------
     public:
          Test(Test &t);                                    //copy
          constructor, reference object as parameter
          --------
 };
Test T1; //object T1 created, default constructor
Test T2=T1;  //copy constructor
//Alternative
 Test T2(T1);
```

Calling the constructor:

Default

- Whenever an object is instantiated, a default constructor is called automatically to reserve the memory for that object.
- A default constructor is called if no user-defined constructor is provided in class definition and it allocates memory for objects.
- It is not possible to pass any parameters to the default constructor and to make it perform or call any other process.

    Account acc1, acc2; //default constructor is called.

User-defined

- The main advantage of user-defined constructor is to initialize the object while it is created.
- A user-defined constructor have following special characteristics
    - ❖ It takes same name as that of class.
    - ❖ Should be declared in the public section of class definition.
    - ❖ They are invoked automatically when the object are created.
    - ❖ May take argument but do not have any return type.

Copy constructor:

- Called using assignment operator or object as arguments automatically.

Parameterized Constructor:

Constructors can be invoked with arguments, just as in the case of functions. The argument list can be specified within braces similar to the argument-list in the function. So a user-defined constructor having arguments is called parameterized constructor. While creating object, we should pass argument for the parameterized constructor. Since C++ allows function overloading, a constructor with arguments can co-exist with another constructor without argument.

```
class Test
{
     private:
          ---------
          ---------
     public:
          Test(int data1)//constructor with parameter
          {
              --------------
          }
          --------
 };
Test t1(4); //4 is passed as parameter
Test t2=5;  //5 is passed as parameter
```

Example:

```
//example of parameterized constructor
#include<iostream.h>
#include<conio.h>
class BankAccount
{
      private:
                  int accno;
                  float bal;
       public:
            BankAccount(int , float );//constructor
            void display()
            {
                  cout<"Account Info";
                  cout<<"\n Account No:"<<accno;
                  cout<<"\n Balance:"<<bal;
            }

};

//definition of parameterized constructor
BankAccount :: BankAccount(int ac, float b)
      {     accno = ac;  bal = b;  }



main()
{
      BankAccount acc1(1024, 5000.0);    //constructor called implicity
      acc1.display();
      BankAccount acc2 = BankAccount(2024, 5000.0); //called explicity
      acc2.display();
      getch();
}
```

Constructor overloading:

A class can have multiple constructors. If more than one constructors are used in a class, it is known as constructor overloading. All the constructors have the same name as the corresponding class, and they differ in terms of number of arguments, data types of argument or both. This makes the creation of object flexible.

```
//constructor overloading
#include <iostream.h>
#include <conio.h>

class Account
{
       private:
            int accno;   float balance;
      public:
            Account()   //constructor1
            { accno=1024,balance=5000.55;}


            Account(int acc)   //constructor2 with one argument
            {
                  accno=acc;
                  balance=0.0;
            }

            Account(int acc, float bal)//constructor3,with two arguments
            {
                  accno=acc; balance=bal;
            }

            void display()
            {
                  cout<<"Account no.="<<accno<<endl;
                  cout<<"Balance="<<balance<<endl;
            }
 };  //end of class definition

 void main()
 {
      Account acc1;  //constructor1
      Account acc2(100);  //constructor2
      Account acc3(200, 8000.50);  //constructor3
      cout<<endl<<"Account information"<<endl;
      acc1.display();
      acc2.display();
      acc3.display();
      cout<<"Account information"<<endl;
      acc1.display();
      acc2.display();
      acc3.display();
   getch();
 } //end of main()
```

Another Example

```
//User-defined copy constructor
#include<iostream.h>
#include<conio.h>
class Code
{
       private:
            int id;
      public:
            Code(){}
            Code(int a) {id=a;}
            Code(Code &x) //reference object argument
            { id = x.id;  }

            int display()
            {     return id; }
}; //end of class definition

void main()
{
  Code p1(55);
  Code p2(p1);  //copy constructor
  Code p4=p2;
  Code p3=p1;
  cout<<"\n Code p1="<<p1.display();
  cout<<"\n Code p2="<<p2.display();
  cout<<"\n Code p3="<<p3.display();
  cout<<"\n Code p4="<<p4.display();
  getch();
}
// All prints 55
```

Destructors

When an object is created the constructed is called automatically. Like wise a class can have another special member function called the destructor. When an object is no longer needed it can be destroyed. Destructor is invoked when an object is destroyed. This function complements the operation performed by any of the constructors. When a program is interrupted using an exit( ) call, the destructor are called for all objects which exist at that time. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde (~).

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible. It may be a good practice to declare destructors in a program since it releases memory space for future use.

```
class Test
{
     private:
          ---------
          ---------
     public:
          Test()//constructor
          {  }
          ~Test()//destructor
          {}
 };
```

Example:

```
//example of destructor
#include<iostream.h>
#include<conio.h>
class Test
{
      public:
      Test()//constructor
      {
      cout<<"\nControl is in constructor";
      }
      ~Test()//destructor
      {
      cout<<"\n Control is in destructor";
      }
};

void main()
{
      Test t; //constructor is called
      cout<<"Function main() terminating...";
      //object t goes out of scope, destructor is called
}
```