Characteristics of Member functions

Member Functions can be

- functions
- Operator
- friend
- static

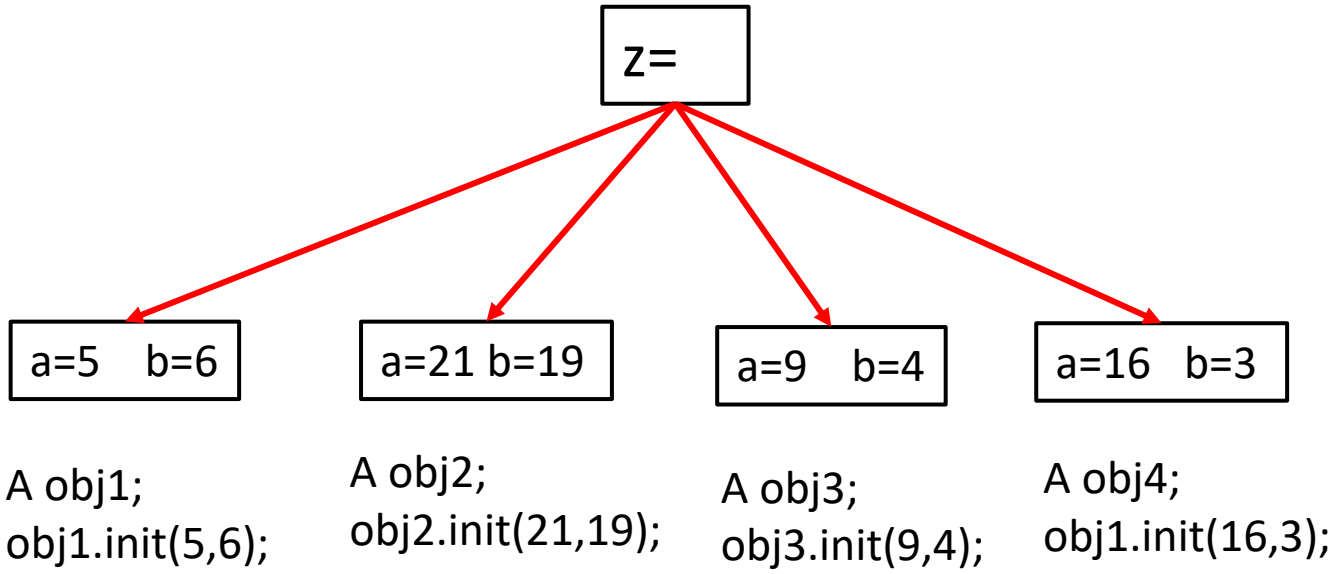
- Special member functions
  - Constructors
  - destructors

The default constructors, destructors, copy constructors, and copy assignment operators are *special member functions*. These functions create, destroy, convert, initialize, and copy class objects.

Static

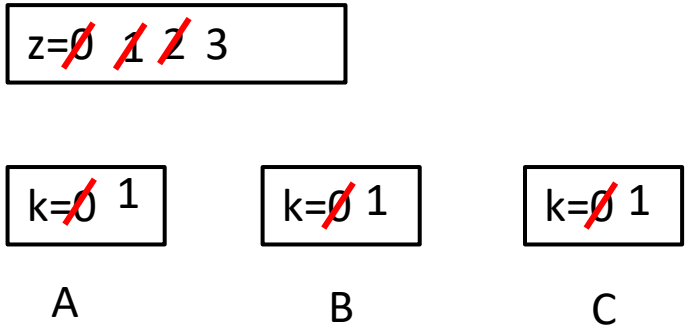
- In C ,static members (variable or functions) are used to restricted to access them with in the file.
- In C++, static keyword to define the class members (data or functions).

```
class A {
    int a,b;          // instance variable
    static int z;     // class variable
    public :
      void init(int x,int y) {
          a=x;b=y;
      }
      void display(){
           cout<<“a=“ <<a
                <<“b=“<<b<<endl;
      }
};
```



- The `static` data members are associated with the `class` and not with any object and stored in static area
- The `static` data member variable must be initialized otherwise the linker will generate an error.

```
class Number
{
   static int z; // static variable
   int k; // non-static variable
   public:
      void zero() {
         k=0;
      }
      void count() {
         ++z;
         ++k;
         cout<<"Value of z = "<<z <<" Address of z = "<<&z<<endl;
         cout<<"Value of k = "<<k <<" Address of k = "<<&k<<endl;
      }
};
int Number :: z=0; // initialization of static member variable
int main()
{
   number A,B,C;
   A.zero(); B.zero(); C.zero();
   A.count(); B.count(); C.count();
   return 0;
}
```



- Similar to static variable, static methods are class methods
- In static methods only static members are accessible , instance members are not accessible because ,***this*** pointer is not available to static methods.
- static methods are called using classname::methodname .

```
class X {
      private:
        int i;
        static int si;
      public:
        void set_i(int arg) { i = arg; }
        static void set_si(int arg) { si = arg; }
        void print_i() {
          cout << "Value of i = " << i << endl;
          cout << "Again, value of i = " << this->i << endl;
        }
        static void print_si() {
          cout << "Value of si = " << si <<endl;
      //  cout << "Again, value of si = " << this->si << endl; // error
        }
};
```

```
int X::si = 77;      // Initialize static data member
int main() {
        X xobj;
        xobj.set_i(11);
        xobj.print_i();
        // static data members and functions belong to the
        // class and can be accessed without using an
        // object of class X
        X::print_si();
        X::set_si(22);
        X::print_si();
        return 0;
}
```

why constructors?

```
class Box {
   double length,breadth,height;
 public:
   void initialize(double l,double b,double h){
      length=l;
      breadth=b;
      height=h;
   }
   double volume(){
      return length*breadth*height;
   }
   double perimeter(){
      return 4*(length+breadth+height);
   }
};
```

```
int main(){
   Box b1,b2;
   b1.initialize(10.0,10.0,10.0);
   cout<<"Area of box1="<<b1.volume()<<endl;
   cout<<"Perimeter of box1="<<b1.perimeter()<<endl;
   cout<<“Area of box 2=“<<b2.volume()<<endl;
   cout<<“Perimeter of box2=“<<b2.perimeter<<endl;
   b2.initialize(15.0,10.0,20.0);
   return 0;
}
```

- Here ,calling of utility operations(area and perimeter) on box objects is valid after initialization.
- We need a mechanism to initialize the object at the time of creation . So , use constructors to create and initialize the object.

Constructor & Destructor

- Constructor is a special member function that is used
  - to ***create an object*** i.e., which allocates memory to the data members of the class in the heap area.
  - ***Initialize the data members during the creation of the object***.
- If the user doesn't specify the constructor in the class ***then the compiler will provide the default constructor.***
- If the user specifies the constructor in the class ***then it overrides the compiler's default constructor.***
- The syntax for constructor is

```
classname(<argumentlist>) {
 /*..*/
}
```

The types of constructors

- default constructors - provided by the compiler.
- Parameterized constructor – constructor with one or more arguments.
- Copy constructor – to create a new object from an existing one by initialization.(A obj2(obj1))
- Copy assignment constructor – assigning/updating a object data by assignment. (obj1=obj2)

Note: Except for parameterized constructor ,the compiler will provide a default constructors for all the above type of constructors

```
class A {
    int a,b;
    public:
        A(){
          cout<<"constructor called and its address = "<<this<<endl;
        }
        ~A() {
          cout<<"Destructor called and its address= "<<this<<endl;
       }
        void init(int x,int y){
              a=x;b=y;
        }
        void display(){
           cout<<"a= "<<a<<"b= "<<b<<endl;
        }
};
```

```
int main(){
    A obj1;
    cout<<"address of obj1="<<&obj1<<endl;
    obj1.init(10,20);
    obj1.display();
     A obj2;
    cout<<"address of obj2="<<&obj2<<endl;
    obj2.init(30,40);
    obj2.display();
    return 0;
}
```

Construction of objects is done in the order specified in the program .But, destruction of objects is done in the reverse order of the construction.

```
constructor called and its address = 0xbe1f5ffb48
address of obj1=0xbe1f5ffb48
a= 10b= 20
constructor called and its address = 0xbe1f5ffb40
address of obj2=0xbe1f5ffb40
a= 30b= 40
Destructor called and its address = 0xbe1f5ffb40
Destructor called and its address = 0xbe1f5ffb48
```

Parameterized constructor

- A constructor that takes one or more parameters is known as parameterized constructor.
- If a class contains a user-defined constructor then the compiler's default constructor is no longer exists.

```
class Box {
   double length,breadth,height;
 public:
   Box(double l,double b,double h){
      length=l;breadth=b;height=h;
   }
   double volume(){
      return length*breadth*height;
   }
   double perimeter(){
      return 4*(length+breadth+height);
   }
};
```

or

```
Box(double l,double b,double h): length(l),breadth(b),height(h){ }
```

```
int main(){
   Box b1(10.0,10.0,10.0);
   Box b2(15.0,10.0,20.0);
   cout<<"Area of box1="<<b1.volume()<<endl;
   cout<<"Perimeter of box1="<<b1.perimeter()<<endl;
   cout<<“Area of box 2=“<<b2.volume()<<endl;
   cout<<“Perimeter of box2=“<<b2.perimeter<<endl;
   return 0;
}
```

Copy constructor

- Copy constructor is needed for initializing the data members of a user-defined datatype(class) from an existing value.
- In call-by-value, make a copy of actual parameter to a formal parameter so, copy constructor is needed.
- In return-by-value , make a copy of the computed value as a return value so, copy constructor is needed.

```
class A {
   public:
     int a,b;
     A(int x,int y):a(x),b(y){
        cout<<"constructor was called="<<this<<endl;
     }
     ~A(){
         cout<<"destructor was called="<<this<<endl;
     }
     A(const A& a1){
      a=a1.a; b=a1.b;
      cout<<"copy constructor was called= "<<this<<endl;
     }
};
void display(A obj){
   cout<<"a value = "<<obj.a<<", b value = "<<obj.b<<endl;
}
int main(){
   A obj1(10,20);
   A obj2(30,40);
   A obj3(obj2);
   display(obj1);
   display(obj2);
   display(obj3);
}
```

Copying obj1 to obj

Copying obj2 to obj

Copying obj3 to obj

```
constructor was called=0xd56fdff9d0
constructor was called=0xd56fdff9c8
copy constructor was called= 0xd56fdff9c0
copy constructor was called= 0xd56fdff9d8
a value = 10, b value = 20
destructor was called=0xd56fdff9d8
copy constructor was called= 0xd56fdff9e0
a value = 30, b value = 40
destructor was called=0xd56fdff9e0
copy constructor was called= 0xd56fdff9e8
a value = 30, b value = 40
destructor was called=0xd56fdff9e8
destructor was called=0xd56fdff9c0
destructor was called=0xd56fdff9c8
destructor was called=0xd56fdff9d0
```

Example 1: constructor overloading

Similar to function overloading, constructors can be overloaded

```
class Box {
   double length,breadth,height;
  public:
   Box() {
       cout<<“zero parameter constructor called”<<endl;
       length=breadth=height=1;
   }
   Box(double l,double b,double h){
       cout<<“three parameter constructor called”<<endl;
       length=l;breadth=b;height=h;
   }
   Box(double l,double b){
       cout<<“two parameter constructor called”<<endl;
       length=l;breadth=b;height=1;
   }
   double volume(){
      return length*breadth*height;
   }
   double perimeter(){
      return 4*(length+breadth+height);
   }
};
```

Zero parameter constructor

three parameter constructor

two parameter constructor

```
int main(){
   Box b1;
   Box b2(10.0,10.0,10.0);
   Box b3(10.0,20.0);
    cout<<"Area of box1="<<b1.volume()<<endl;
   cout<<"Perimeter of box1="<<b1.perimeter()<<endl;
   cout<<“Area of box 2=“<<b2.volume()<<endl;
   cout<<“Perimeter of box2=“<<b2.perimeter<<endl;
   cout<<“Area of box 3=“<<b3.volume()<<endl;
   cout<<“Perimeter of box3=“<<b3.perimeter<<endl;
   return 0;
}
```

Example 2: constructor overloading

```
class Point{
   public:
     int x,y;
     Point(int x1,int y1):x(x1),y(y1){}
};

class Rectangle{
   Point left_top, right_bottom;
   public:
         Rectangle(Point p1,Point p2):left_top(p1),right_bottom(p2){}
         Rectangle(Point p1,int h,int w):left_top(p1),right_bottom(Point(p1.x+w,p1.y+h)){}
         Rectangle(int h,int w):left_top(Point(0,0)),right_bottom(Point(h,w)){}
         double area(){
            return (right_bottom.x-left_top.x)*(right_bottom.y-left_top.y);
         }
};

int main(){
   Point p1(2,5),p2(8,10);
   Rectangle r1(p1,p2),r2(p1,5,6),r3(5,6);
   cout<<"area="<<r1.area()<<endl;
   cout<<"area="<<r2.area()<<endl;
   cout<<"area="<<r3.area()<<endl;
   return 0;
}
```

Friend function

```
class A{
        int a,b;
    public:
         A(int x,int y):a(x),b(y){}
};
void display(A& obj){
        cout<<"a value="<<obj.a<<endl;
        cout<<"b value="<<obj.b<<endl;
}
int main(){
        A obj1(10,20);
        display(obj1);
        return 0;
}
```

Error , because private members are not accessible outside the class.

Global function (non-member function)

```
class A{
        int a,b;
    public:
        A(int x,int y):a(x),b(y){}
        friend void display(A&);
};
void display(A& obj){
         cout<<"a value="<<obj.a<<endl;
         cout<<"b value="<<obj.b<<endl;
}
int main(){
        A obj1(10,20);
        display(obj1);
        return 0;
}
```

friend function declaration

Why?

To provide access to ***private and protected members of a class*** to global functions(non-member functions) or other class members use friend keyword.

- It is useful for a class to grant member-level access to functions that aren't members of the class, or to all members in a separate class. These non member functions and classes are known as *friends*, marked by the **friend** keyword.

- Only the class implementer can declare who its friends are. A function or class can't declare itself as a friend of any class.

- A **friend** function is a function that isn't a member of a class but has access to the class's private and protected members.
- Friend functions aren't considered class members; they're normal external functions that are given special access privileges and they can be called without an object of the class(except they are members of another class)

Example 2:

```
class Node; // forward declaration
class List{
        Node *head;
        Node *tail;
    public:
        List():head(0),tail(0){}
        void display();
        void append(Node*);
};
class Node{
        int data;
        Node *next;
    public:
        Node(int i):data(i){}
        friend void List::display();
        friend void List::append(Node*);
};
```

```
void List::display(){
        Node *p=head;
        while(p){
            cout<<p->data<<" --";
            p=p->next;
        }
}
void List::append(Node *p){
        if(!head) head=tail=p;
        else{
            tail->next=p;
            tail=tail->next;
        }
}
int main(){
        List l;
        Node n1(10),n2(20),n3(30);
        l.append(&n1);
        l.append(&n2);
        l.append(&n3);
        l.display();
        return 0;
}
```