Introduction to C++

- Destructor
- this keyword
- Inline Functions
- Friend Function
- Friend Class
- Two classes having same friend

Destructor

Destructor is a special member function which deletes an object.

- A destructor function is called automatically when the object goes out of scope. Whereas constructors are designed to initialize a class, destructors are designed to help clean up.
- Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.
- Destructors are called in back order relatively to the order of objects created. The first created object is destroyed last. And the last created object is destroyed first.

When destructor call:

- when program ends
- when a delete operator is called

Features of destructor:

- The same name as the class but is preceded by a tilde (~)
- no arguments and return no values

Syntax of Destructor is:

```
~classname()
{ ......
}
```

Destructor

Simple Example using Destructor:

```
#include<iostream>
using namespace std;
class Student
{
public:
int rollno;
Student()
{
        cout<<"constructor called"<<endl;
}
~Student()
{
        cout<<"destructor called";
}
};
int main()
{
        Student s1;
        Student s2;
}
```

Destructor

Example of Destructor:

```
#include<iostream>
using namespace std;
class Student
{public:
int rollno;
Student(int x)
{           rollno=x;
            cout<<"constructing "<<rollno<<endl;
}
~Student()
{           cout<<"destructing "<<rollno<<endl;
} };
int main()
{           Student s1(1); Student s2(2);
}
```

O/P:constructing 1
constructing 2
destructing 2
destructing 1

As we can see that Destructors are called in back order relatively to the order of objects created.

this keyword

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.

The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. 'this' pointer is not available in static member functions as static member functions can be called without any object (with class name).

Following are the situations where 'this' pointer is used:

1) When local variable's name is same as member's name.
2) inside a member function, this may be used to refer to the invoking object.

this keyword

Program using this when local variable's name is same as member's name :

```
#include<iostream>
using namespace std;
class Student
{ int rollno;
 string name;
 public:
 Student(int rollno,string name);
 void getStudent();
 };
 Student::Student(int rollno,string name)
 {this->rollno=rollno;
 this->name=name;}
 void Student::getStudent()
 {cout<<"rollno is "<<rollno<<endl;
  cout<<"name is "<<name<<endl;}
  int main()
  {
  Student s(6,"Vivek");
  s.getStudent();
  }
```

this keyword

Program where this is used to refer to the invoking object.

```
#include <iostream>
using namespace std;
class Box
{
  private:
    float length;     // Length of a box
    float breadth;    // Breadth of a box
    float height;     // Height of a box
  public:
    Box(float l,float b,float h)
    {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
    }
    float Volume()
    {
      return length * breadth * height;
    }
```

this keyword

```
int compare(Box box)
     {
       return this->Volume() > box.Volume();
     }
 };
int main()
{
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2
   if(Box1.compare(Box2))
   { cout << "Box2 is smaller than Box1" <<endl;  }
   else
   { cout << "Box2 is equal to or larger than Box1" <<endl; }
}
```

O/P:

Constructor called.

Constructor called.

Box2 is equal to or larger than Box1

Inline functions

Inline function is powerful concept in C++ programming language. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.
The keyword inline is used as a function specifier in function declaration.inline keyword is the hint to the compiler that inline substitution of the function body is to be preferred to the usual function call implementation.
The inline specifier can be used either as a member of the class or a global function.

- To inline a function, place the keyword inline before the function name and define the function before any calls are made to the function.
- A function definition in a class definition is an inline function definition, even without the use of the inline specifier.

Advantage of Inline Function:
The main advantage of inline function is it make the program faster. Inline functions are used to improve performance of the application.

Syntax:
inline return_type function_name()
{ //function body }

Inline functions

There are two ways to make the functions to be inline. The first one consists in simple definition of member function in the body of class declaration:

Example:

```
#include<iostream>
using namespace std;
class A
{
        int number;
        public:
        void getNumber()  //this is an inline function.
{
        cout<<"enter a number"<<endl;
        cin>>number;
        cout<<"Number is "<<number;
}};
int main()
{
        A a1;
        a1.getNumber();
}
```

Inline functions

Another possibility is to use inline keyword with definition of function. In this case, we have to declare function in the same way as we declare a simple function:

```
#include<iostream>
using namespace std;
class A
{
        int number;
        public:
        void getNumber();
        void display();
};
inline void A::getNumber()
{
        cout<<"enter a number";
        cin>>number;
}
inline void A::display()
{
        cout<<"Number is "<<number;
}
```

Inline functions

```
inline void hello()
 {
 cout<<"hello";
 }
int main()
{
            A a1;
            a1.getNumber();
            a1.display();
            hello();
}
```

However, once the program is compiled, the call to getNumber(),display(),hello(); will be replaced by the code making up the function.

Friend function

In C++ a friend function that is a "friend" of a given class is allowed access to private and protected data in that class.

- A function can be made a friend function using keyword friend. Any friend function is preceded with friend keyword. The declaration of friend function should be made inside the body of class (can be anywhere inside class either in private or public section) starting with keyword friend.
- Even though the prototypes for friend functions appear in the class definition, friends are not member functions.
- Friend function declaration can appear anywhere in the class,but a good practice would be where the class ends.

Why use friend function?
We can't access private or protected data member of any class, to access private and protected data member of any class we need a friend function.

Syntax:

```
class class_name
{ ......
friend returntype function_name(arguments);
}
```

Friend function

Example of friend function:

```
#include<iostream>
using namespace std;
class A
{       int x,y;
        public:
                void get()
                {       cout<<"Enter 2 values";
                        cin>>x>>y;
                }
                friend float meanValue(A ob);
};
float meanValue(A ob)
{       return (ob.x+ob.y)/2;
}
int main()
{       A a1;
        a1.get();
        cout<<"Mean value is "<<meanValue(a1);
}
```

Here, friend function meanValue() is declared inside A class. So, the private data can be accessed from this function.

Friend class

Just like functions are made friends of classes, we can also make one class to be a friend of another class. Then, the friend class will have access to all the private members of the other class.
When a class is made a friend class, all the member functions of that class becomes friend function.
If B is declared friend class of A then, all member functions of class B can access private and protected data of class A but, member functions of class A can not access private and protected data of class B.

Syntax:

```
class A
{
friend class B; //class B is a friend of class A
……..
}
class B
{
……..
}
```

Friend class

Example of friend class:

```
#include <iostream>
using namespace std;
class A
{           private:
            int x;
            public:
            A()
            {    x=10;    }
            friend class B;
};
class B
{          private:
            int b;
            public:
            void show(A &a1)
            {
             cout<<"A::a= "<<a1.x;
            }
};
int main()
{           A a;
            B b;
            b.show(a);
}
```

Two classes having same friend

Two Classes having the same friend:

A non-member function may have friendship with one or more classes.When a function has declared to have friendship with more than one class,the friend classes should have forward declaration.

Syntax:

```
class Second;   //forward declaration
class First
{
……
friend return_type func_name(class First,class Second);
};
class Second
{
…..
friend return_type func_name(class First,class Second);
};
```

Two classes having same friend

Example:

```
#include<iostream>
using namespace std;
class Second;
class First
{ private:
            int x;
  public:
            void getData();
            friend int sum(First,Second);
};
class Second
{ private:
            int y;
  public:
            void getData();
            friend int sum(First,Second);
};
void First::getData()
{ cout<<"enter the value of x";
  cin>>x;}
```

Two classes having same friend

```
void Second::getData()
{
  cout<<"enter the value of y";
  cin>>y;
}
int sum(First one,Second two)
{
  int z;
  z=one.x+two.y;
  return z;
}
int main()
{
  First a;
  Second b;
  a.getData();
  b.getData();
  cout<<"sum of two data variables="<<sum(a,b);
}
```

Discussions