OBJECT ORIENTED PROGRAMMING LAB

Lab Manual # 06

Static & const Keyword

Instructor: Iqra Rehman

Semester Spring, 2024

Course Code: CL1004

Fast National University of Computer and Emerging Sciences Peshawar

Department of Computer Science

OBJECT ORIENTED PROGRAMMING LANGUAGE

Table of Contents

C++ static Keyword

In C++, static is a keyword or modifier that belongs to the type not instance. So, instance is not required to access the static members. Static keyword in C++ is use to give special characteristics to an element. Static elements are allocated storage only once in a program lifetime in static storage area. And they have a scope till the program lifetime.

In C++, static can be field (variable), method and objects.

Advantage of C++ static keyword

Memory efficient: Now we don't need to create instance for accessing the static members, so it saves memory.

Moreover, it belongs to the type, so it will not get memory each time when instance is created.

Static Variables inside Functions

Static variables when used inside function are initialized only once, and then they hold there value even through function calls. These static variables are stored on static storage area , not in stack.

```
#include <iostream>
using namespace std;
void counter()
{
    static int count=0;
    cout << count++ << endl;
}
int main()
{
    for(int i=0;i<5;i++)
    {
        counter();
    }
return 0;
}

/*
Output:
0
1
2
3
4
*/
```

If we do not use static keyword, the variable count, is reinitialized everytime when counter() function is called, and gets destroyed each time when counter() functions ends. But, if we make it static, once initialized count will have a scope till the end of main() function and it will carry its value through function calls too.

If you don't initialize a static variable, they are by default initialized to zero.

C++ Static Field

- A field which is declared as static is called static field.
- Unlike instance field which gets memory each time whenever you create object, there is only one copy of static field created in the memory.
- It is shared to all the objects.
- It is used to refer the common property of all objects such as rateOfInterest in case of Account, companyName in case of Employee etc.
- Let's see the simple example of static field in C++.

C++ static field example

```
#include <iostream>
using namespace std;
class Account {
   public:
       int accno;     //data member (also instance variable)
       string name;   //data member(also instance variable)
       static float rateOfInterest;
       Account(int accno, string name)
        {
            this->accno = accno;
            this->name = name;
        }
       void display()
        {
            cout<<accno<< " "<<name<< " "<<rateOfInterest<<endl;
        }
};
// Initialize static member of class Account
float Account::rateOfInterest=6.5;
int main(void) {
   Account a1 =Account(201, "Kashif");//creating an object of Employee

   Account a2= Account(202, "Amir"); //creating an object of Employee
```

```
    a1.display();
    a2.display();

    return 0;
}
/*
Output
201 Kashif 6.5
202 Amir 6.5
*/
```

Uses of static class data

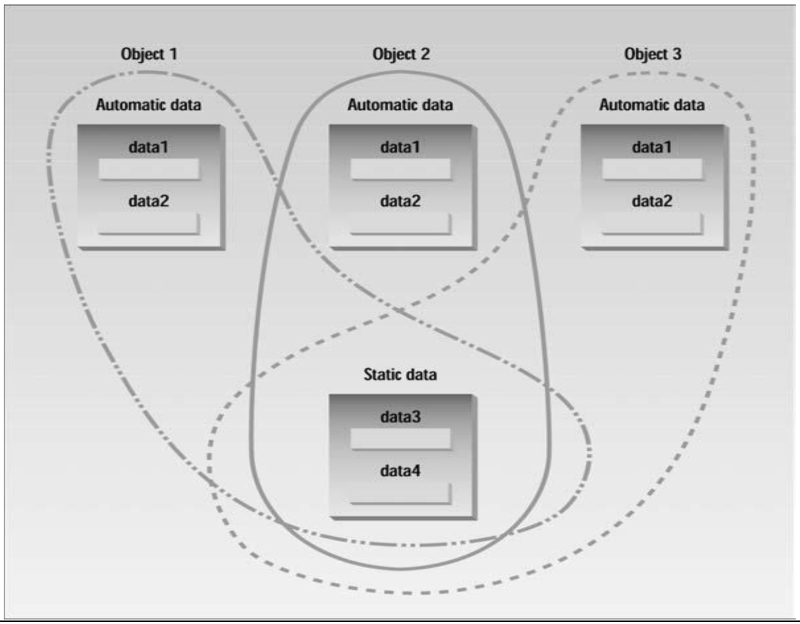
Why would you want to use static member data ?

An example, suppose an object needed to know how many other objects of its class were in the program. In road-racing game**, for example**, a race car might want to know how many other cars are still in the race. In this case a static variable count could be included as a member of the class. All the objects would have access to this variable. It would be the same variable. It would be the same variable for all of them; they would all see the same count.

C++ static field example: Counting Objects

```
#include <iostream>
using namespace std;
class Count
{
private:
static int count; //only one data item for all objects
//note: "declaration" only!
public:
Count() //increments count when object created
{
    count++;
       //cout<<count;
}
int getcount() //returns count
{
return count;
}
};

//----------------------------------------------------------------
int Count::count = 0; //*definition* of count
////////////////////////////////////////////////////////////////
int main()
{
```



```
Count c1, c2, c3; //create three objects
cout << "Count is "<<  c1.getcount() << endl; //each object
cout << "Count is " << c2.getcount() << endl; //sees the
cout << "Count is " << c3.getcount() << endl; //same value
return 0;
}
```

The class Count in this example has one data item, count, which is type static int. The constructor for this class causes count to be incremented. In main() we define three objects of class Count. Since the constructor is called three times, count is incremented three times. Another member function, getcount(), returns the value in count. We call this function from all three objects, and—as we expected—each prints the same value. Here's the output:

- count is 3 <-------static data
- count is 3
- count is 3

If we had used an ordinary automatic variable—as opposed to a static variable—for count, each constructor would have incremented its own private copy of count once, and the output would have been

- count is 1 <------automatic data
- count is 1
- count is 1

Static class variables are not used as often as ordinary non-static variables, but they are important in many situations. Figure in next slide shows how static variables compare with automatic variables.

```
#include <iostream>
using namespace std;
class Account {
   public:
       int accno; //data member (also instance variable)
       string name;
       static int count;
       Account(int accno, string name)
        {
            this->accno = accno;
            this->name = name;
            count++;
        }
       void display()
        {
            cout<<accno<<" "<<name<<endl;
        }
};
int Account::count=0;
int main(void) {

    Account a1 =Account(201, "Ali"); //creating an object of Account
    Account a2=Account(202, "Saad");
    Account a3=Account(203, "Sharjeel");

    a1.display();
    a2.display();
    a3.display();
    cout<<"Total Objects are: "<<Account::count;
    return 0;
}
/*
Output
201 Ali
202 Saad
203 Sharjeel
Total Objects are: 3
*/
```

We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

Let us try the following example to understand the concept of static data members –

```
#include <iostream>

using namespace std;

class Box {
   public:
      static int objectCount;

      // Constructor definition
      Box(double l = 2.0, double b = 2.0, double h = 2.0) {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;

         // Increase every time object is created
         objectCount++;
      }
      double Volume() {
         return length * breadth * height;
      }

   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

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

   // Print total number of objects.
   cout << "Total objects: " << Box::objectCount << endl;

   return 0;
}
/*
Output
When the above code is compiled and executed, it produces the
following result -
Constructor called.
Constructor called.
```

```
Total objects: 2 */
```

Static Function Members

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Let us try the following example to understand the concept of static function members –

```
#include <iostream>

using namespace std;

class Box {
   public:
      static int objectCount;

      // Constructor definition
      Box(double l = 2.0, double b = 2.0, double h = 2.0) {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;

         // Increase every time object is created
         objectCount++;
      }
      double Volume() {
         return length * breadth * height;
      }
      static int getCount() {
         return objectCount;
      }

   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;
int main(void) {
   // Print total number of objects before creating object.
   cout << "Inital Stage Count: " << Box::getCount() << endl;

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

   // Print total number of objects after creating object.
   cout << "Final Stage Count: " << Box::getCount() << endl;

   return 0;
}
```

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

```
Inital Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
```

Static Objects

Static objects are declared with the keyword static. They are initialized only once and stored in the static storage area. The static objects are only destroyed when the program terminates i.e. they live until program termination.

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

class Abc {
      int i;
public:
      Abc() {
            i = 0;
            cout << "  constructor  ";
      }
      ~Abc() {
            cout << "  destructor  ";
      }
}; //class ends here

//function: call to this function will create object of class ABC
void f() {
//    Abc obj;
      static Abc obj;
}

int main() {

     int x = 0;
     if (x == 0) {
          f();
          cout<<" end of if  ";
     }

     cout << "  END  ";
}

/*
Output:
 constructor  end of if   END   destructor
*/
```

You must be thinking, why was the destructor not called upon the end of the scope of if condition, where the reference of object obj should get destroyed. This is because object was static, which has scope till the program's lifetime, hence destructor for this object was called when main() function exits.

Const Keyword

It is the const keywords used to define the constant value that cannot change during program execution. It means once we declare a variable as the constant in a program, the variable's value will be fixed and never be changed. If we try to change the value of the const type variable, it shows an error message in the program.

Const variable

It is a const variable used to define the variable values that never be changed during the execution of a program. And if we try to modify the value, it throws an error.

```
     const data_type variable_name;
```

Example:

```
#include <iostream>

#include <conio.h>

using namespace std;
```

```
int main ()

{

     // declare the value of the const

     const int num = 25;

     num = num + 10;

     return 0;

}


/*

Output:

It shows the compile-time error because we update the assigned value of
the num 25 by 10.

*/
```

const Data Member

Data members are like the variable that is declared inside a class, but once the data member is initialized, it never changes, not even in the constructor or destructor. The constant data member is initialized using the const keyword before the data type inside the class. The const data members cannot be assigned the values during its declaration; however, they can assign the constructor values.

Example:

```
#include <iostream>

using namespace std;

// create a class ABC

class ABC

{
```

```
public:

 // use const keyword to declare const data member

const int A;

// create class constructor

ABC ( int y) : A(y)

{

cout << " The value of y: " << y << endl;

}

};

int main ()

{

ABC obj( 10); // here 'obj' is the object of class ABC

cout << " The value of constant data member 'A' is: " << obj.A << endl;

// obj.A = 5; // it shows an error.

// cout << obj.A << endl;

return 0;

}


/*

Output:

The value of y: 10

The value of constant data member 'A' is: 10

*/
```

const Member Functions

- A const member function guarantees that it will never modify any of its class's member data.

- A function is made into a constant function by placing the keyword const after the declarator but before the function body.
- Member functions that do nothing but acquire data from an object are obvious candidates for being made const, because they don’t need to modify any data.

Example:

```
class aClass {

private:

     int alpha;

public:

     void nonFunc()                 //non-const member function

     { alpha = 99; }                //OK


     void conFunc() const           //const member function

     { alpha = 99; }                //ERROR: can’t modify a member

};
```

The non-const function nonFunc() can modify member data alpha, but the constant function conFunc() can’t. If it tries to, a compiler error results.

const Member Function Arguments

If an argument is passed to an ordinary function by reference, and you don’t want the function to modify it, the argument should be made const in the function declaration (and definition). This is true of member functions as well. For example:

```
Distance Distance::add_dist(const Distance& d2) const

{
    Distance temp;                           //temporary variable

    // feet = 0;                             //ERROR: can’t modify this

    // d2.feet = 0;                          //ERROR: can’t modify d2

    temp.inches = inches + d2.inches;        //add the inches

    if(temp.inches >= 12.0)                  //if total exceeds 12.0,

    {                                        //then decrease inches

            temp.inches -= 12.0;             //by 12.0 and

            temp.feet = 1;                   //increase feet

    }                                        //by 1

    temp.feet += feet + d2.feet;             //add the feet

    return temp;

}
```

Example : Distance

```
#include <iostream>

using namespace std;


class Distance {


private:

    int feet;

    float inches;


public:

	//constructor (no args)
	Distance() : feet(0), inches(0.0) { }

	//constructor (two args)
	Distance(int ft, float in) : feet(ft), inches(in) {
	}

	//get length from user
	void getdist() {

		cout << "\nEnter feet: ";
		cin >> feet;

		cout << "Enter inches: ";
		cin >> inches;
	}

	//display distance
	void showdist() const {
		cout << feet << "\'-" << inches << '\"';
	}

	Distance add_dist(const Distance&) const; //const member function with
const function argument

};
```

```
Distance Distance::add_dist(const Distance &d2) const

{

     Distance temp;                //temporary variable

     // feet = 0;                  //ERROR: can’t modify this

     // d2.feet = 0;               //ERROR: can’t modify d2


     temp.inches = inches + d2.inches;     //add the inches


     if (temp.inches >= 12.0)                        //if total exceeds 12.0,

     {
     //then decrease inches

             temp.inches -= 12.0;                    //by 12.0 and

             temp.feet = 1;                                  //increase feet

     }
     //by 1


     temp.feet += feet + d2.feet;          //add the feet


     return temp;


}


int main() {

     Distance dist1, dist3;                          //define two lengths

     Distance dist2(11, 6.25);             //define, initialize dist2
```

```
    dist1.getdist();                              //get dist1 from user

    dist3 = dist1.add_dist(dist2);        //dist3 = dist1 + dist2

    //display all lengths
    cout << "\ndist1 = "; dist1.showdist();
    cout << "\ndist2 = "; dist2.showdist();
    cout << "\ndist3 = "; dist3.showdist();
    cout << endl;
    return 0;
}
/*
Output:
Enter feet: 44
Enter inches: 3

dist1 = 44'-3"
dist2 = 11'-6.25"
dist3 = 55'-9.25”
*/
```

const Objects

we can apply const to variables of basic types such as int to keep them from being modified. In a similar way, we can apply const to objects of classes. When an object is declared as const, you can’t modify it. It follows that you can use only const member functions with it, because they’re the only ones that guarantee not to modify it.

```
    const Distance football(300, 0);

    football.getdist();                   //ERROR: getdist() not const
```

References

https://beginnersbook.com/2017/08/cpp-data-types/

http://www.cplusplus.com/doc/tutorial/basic_io/

https://www.w3schools.com/cpp/default.asp

https://www.javatpoint.com/cpp-tutorial

https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/?ref=lbp

https://www.programiz.com/

https://ecomputernotes.com/cpp/

https://www.studytonight.com/cpp/static-keyword.php