Logging in From a Linux System or Localhost
4 days 42 min ago

Auto response
4 days 6 hours ago

Re: final issue.
4 days 22 hours ago

Re: Rocky, thank you for this
4 days 23 hours ago

Learning C/C++ Step-By-Step - Page 13

Submitted by ganesh35 (Contact Author) (Forums) on Wed, 2009-01-07 18:45. ::

13. Step-by-Step C/C++ --- C++ Programming - OOPs

OOP (Object Oriented Programming) in C++

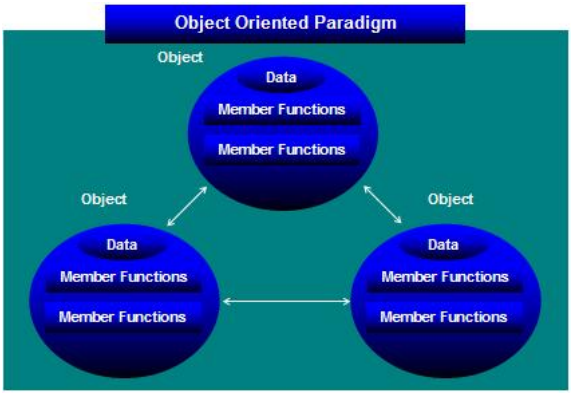
1. Object Oriented Paradigm
2. Characteristics of Object-Oriented Language
   - Objects
   - Classes
   - Data abstraction
   - Data encapsulation
   - Inheritance
   - Polymorphism
   - Dynamic binding
   - Message passing
3. History of C++
4. Classes and Objects
5. Member functions defined outside the class
6. Array of Objects
7. Objects as Arguments
8. Returning Objects from functions
9. Constructor
10. Destructors
11. Constructor Overloading
12. Static Class Data
13. Static Member Functions
14. Friend Functions

1. Object Oriented Paradigm

The basic idea behind the Object Oriented Paradigm is to combine into a single unit of both data and the functions that operate on that data. Such a unit is called an object.

Through this method we cannot access data directly. The data is hidden, so, is safe from
Accidental alteration. Data and its functions are said to be encapsulated into a single entity. Data encapsulation and data hidings are key terms in the description of object-oriented language.

A C++ program typically consists of a number of objects, which communicate with each other by calling one another's member functions. The organization of a C++ program is shown in this figure.



2. Characteristics of Object-Oriented Language

Here are few major elements of Object-Oriented languages.

- Objects
- Classes
- Data abstraction
- Data encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

Objects
Object is an instance of a class. Combining both data and member functions. Objects are the basic run-time entities in an object-oriented system.

Classes
A template or blueprint that defines the characteristics of an object and describes how the object should look and behave.

Data Abstraction
Identifying the distinguishing characteristics of a class or object without having to process all the information about the class or object. When you create a class — for example, a set of table navigation buttons — you can use it as a single entity instead of keeping track of the individual components and how they interact.

Data Encapsulation
An object-oriented programming term for the ability to contain and hide information about an object, such as internal data structures and code. Encapsulation isolates the internal complexity of an object's operation from the rest of the application. For example, when you set the Caption property on a command button, you don't need to know how the string is stored.

Inheritance
An object-oriented programming term. The ability of a subclass to take on the characteristics of the class it's base
the subclass on which it is based inherits those characteristics.
To inherit the qualities of base class to derived class.

Polymorphism
An object-oriented programming term. The ability to have methods with the same name, but different content, for
run time by the class of the object. For example, related objects might both have Draw methods. A procedure, pa
method without needing to know what type of object the parameter is.

Dynamic Binding
Dynamic refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Message Passing
An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language therefore involves the following basic steps:

1. Creating classes that define objects and their behavior.
2. Creating objects from class definitions.
3. Establishing communication among objects.

3. History of C++

Year	Language	Developed by	Remarks
1960	ALGOL	International Committee	Too general, Too abstract
1963	CPL	Cambridge University	Hard to learn, Difficult to implement
1967	BCPL	Martin Richards	Could deal with only specific problems
1970	B	Ken Thompson AT & T Bell Labs	Could deal with only specific problems
1972	C	Dennis Ritchie AT & T Bell Labs	Lost generality of BCPL and B restored
Early 80's	C++	Bjarne Stroustrup AT & T	Introduces OOPs.

C++ is an object-oriented programming language. Initially named 'C with Classes', C++ was developed by **Bjarne Stroustrup** at AT & T Bell laboratories in Murry Hill, New Jersey, USA, in the early eighties.

Stroustrup, an admirer of Simula67 (an OOP language) and a strong supporter of C, wanted to combine the best of both languages and create a more power

and elegance of C. The result was C++.

C++ is a truly Object Oriented Language, So. It must be a collection of classes and objects.

4. Classes and Objects

A class is a way to bind the data and its associated functions together. It allows the data to be hidden, if necessary, from external use. When defining a class, we are creating a new abstract data type that can be treated like any other built-in data type. Generally, a class specification has two parts:

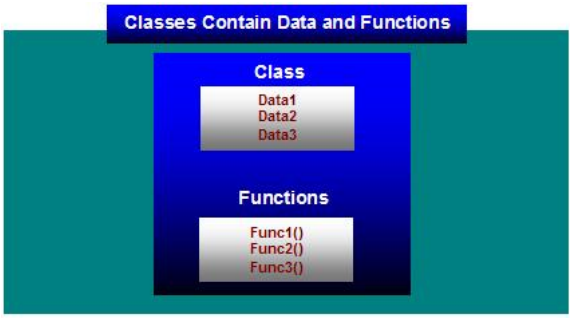
1. Class declaration
2. Class function definition

The declaration specifies the type and scope of both data and member functions of class. Where as definition specifies the executable code of the function.

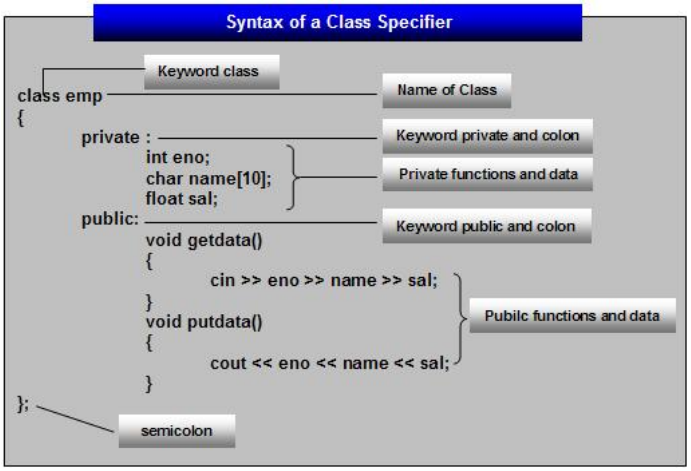
The general form of a class declaration is:

```
class class_name
{
     private:
          variable declarations;
          function declarations;
     public:
          variable declarations;
          function declarations;
};
```

The class declaration is similar to struct declaration. The key word class specifies that the data and functions be of private by default. Where as a struct key word specifies that the data and functions be of public by default. The keywords private and public are known as visibility labels.



Here is an example class to implement an employee class.



The following is the complete program of emp class.

```
// Program to accept and display employee information
#include <iostream>
using namespace std;
class emp                                          // class definition
{
            private :                                   // private data, functions
                        int eno;
                        char name[10];
                        float sal;
            public :                                    // public data, functions
                        void getdata()
                        { cin >> eno >> name >> sal;  }
                        void putdata()
                        { cout << eno << name << sal;  }
};
int main()
{
            emp e;
            e.getdata();
            e.putdata();
            return 0;
}
```

5. Member functions defined outside the class

There is a possibility to define member functions outside of the class using scope resolution operator (::).

```
// Program to accept and display employee information
#include <iostream>
using namespace std;
class emp                                               // class definition
{
            private :                           // private data, functions
                        int eno;
                        char name[10];
                        float sal;
            public :                                        // public data, functions
                        void getdata();
                        void putdata();
};
void emp::getdata()
{   cin >> eno >> name >> sal;  }
void emp::putdata()
{   cout << eno << name << sal; }
int main()
{
            emp e;
            e.getdata();
            e.putdata();
            return 0;
}
```

6. Array of Objects

C++ compiler also supports array of objects.
Below example illustrates the advantage of Objects using arrays.

```
// Program to accept and display employee information
#include <iostream>
using namespace std;
class emp                                               // class definition
{
            private :                           // private data, functions
                        int eno;
                        char name[10];
                        float sal;
            public :                                        // public data, functions
                        void getdata()
                        { cin << eno << name << sal;  }
                        void putdata()
                        { cout >> eno >> name >> sal;  }
};

int main()
{
            emp e[10];                      // declaration of array of objects
            for(i = 0; i <10; i++)        // accessing objects properties and methods
                        e[i].getdata();
           for(i = 0; i< 10; i++)
                        e[i].putdata();
           return 0;
}
```

7. Objects as Arguments

Passing Objects to functions is similar to passing structures, arrays to functions. The following program demonstrates how objects passed to functions.

```
// Program to accept and display employee information
#include <iostream>
using namespace std;
class emp                                               // class
definition
{
            private :                           // private data,
functions
                        int eno;
                        char name[10];
                        float sal;
            public :                                        // public
data, functions
                        void getdata()
                        { cin >> eno >> name >> sal;  }
                        void putdata()
                        { cout << eno << name << sal;  }
};
void operate(emp t);
int main()
{

            emp e;
            operate(e);
}
void operate(emp t)
{
            t.getdata();
            t.putdata();
            return 0;
}
```

8. Returning Objects from functions

We saw objects being passed as arguments to functions, now we will discuss about how to return objects from functions.

```
// Program to accept and display employee information
#include <iostream>
using namespace std;
class emp                                          // class definition
{
            private :                         // private data, functions
                        int eno;
                        char name[10];
                        float sal;
            public :                                      // public data, functions
                        void getdata()
                        { cin >> eno >> name >> sal;  }
                        void putdata()
                        { cout << eno << name << sal;  }
};
emp get();
void put(emp t);
int main()
{

            emp e;
            e = get();
            put(e);
            return 0;
}
emp get()
{
            emp t;
            t.getdata();
            return t;
}
void put(emp t)
{
            t.putdata();
}
```

9. Constructor

The following example shows two ways to give values to the data items in an object. Sometimes, however, it's convenient if an object can initialize itself when it's first created, without the need to make a separate call to a member function.

Automatic initialization is carried out using a special member function called a constructor. A **constructor** is a member function that is executed automatically whenever an object is created.

```
// Program to accept and display employee information using constructors
#include <string.h>
#include <iostream>
using namespace std;
class emp                              // class definition
{
      private :                        // private data, functions
           int eno;
           char name[10];
           float sal;
      public    :                       // public data, functions
            emp() { ; }                 // constructor without arguments
            emp(int teno, char tname[10], float tsal) // constructor with arguments
            {
                 eno = teno;
                 strcpy(name, tname);
                 sal = tsal;
             }
             void getdata()
             { cin >> eno >> name >> sal;  }
             void putdata()
             { cout << eno << name << sal << endl;  }
};

int main()
{
            emp e1(1001, "Magic", 6700.45);
            emp e2;
            e2.getdata();
            e1.putdata();
            e2.putdata();
            return 0;
}
```

The above example program accepts values in two ways using constructors and using member functions. An object, whenever it was declared it automatically initialized with the given values using constructors. Where as object **e2** is accessible by its member function only.

One more example to distinguish the use of constructor.

```
//  Objects represents a counter variable
#include <iostream>
using namespace std;
```

```
class counter
{
            private :
            int count;                                  // variable count
            pubilc :
                  counter()           { count = 0; }    // constructor
                        void inc_count() { count++; }   // increment count
                        int get_count()   { return count; }  // return count
};
int main()
{
            counter c1, c2;                     // define and initialize
            cout << "\nC1 = " << c1.get_count();       // display
            cout << "\nC2 = " << c2.get_count();
            c1.inc_count();                             // increment c1
            c2.inc_count();                             // increment c2
            c2.inc_count();                             // increment c2
            cout << "\nC1 = " << c1.get_count();       // display again
            cout << "\nC2 = " << c2.get_count();
            return 0;
}
```

A constructor has the following characteristics.

- Automatic initialization
- Return values were not accepted
- Same name as the class
- Messing with the format

10. Destructors

A destructor has the same name as the constructor (which is the same as the class name) but preceded by a tilde:

```
// Demonstration of a destructor
#include <iostream>
using namespace std;
class temp
{
            private :
                        int data;
            public :
                        temp() { data = 0;  }                   // Constructor (same
name as class)
                        ~temp() { }                                 // destructor
(same name with tilde)
}
int main()
{
            temp t;
            return 0;
}
```

11. Constructor Overloading

The ability to have functions with the same name, but different content, for related class. The procedure to use is determined at run time by the class of the object.

```
// Demonstration of a constructor overloading
#include <iostream>
using namespace std;
class ttime
{
       private :
               int hh, mm, ss;
       public :
               ttime() {hh = 0; mm = 0; ss = 0; } // Constructor with initialization
               ttime(int h, int m, int s)         // Constructor with 3 arguments
               {
                    hh = h; mm = m ; ss = s;
               }
               ttime(int h, int m)               // Constructor with 2 arguments
               {
                    hh = h; mm = m; ss = 0;
               }
               ttime(int h)                      // Constructor with 1 argument
               {
                     hh = h; mm = 0; ss = 0;
                }
                ~ttime() { }
                void get_time()
                {
                      cin >> hh >> mm >> ss;
                }
                void put_time()
                {
                      cout << endl << hh << "  " << mm <<"  " <<  ss;
                }
};
int main()
{
            ttime t1, t2(12, 12, 12), t3(4, 5), t4(11);       // Calling constructors
            t1.get_time();
            t1.put_time();
            t2.put_time();
            t3.put_time();
            t4.put_time();
            return 0;
}
```

12. Static Class Data

If a data item in a class is defined as **static,** then only one such item is created for the entire class, no matter how many objects there are. A static data item is

useful when all objects of the same class must share a common item of information. A member variable defined as **static** has similar characteristics to a normal static variable: It is visible only within the class, but its lifetime is the entire program.

```
// Demonstration of a static data
#include <iostream>
using namespace std;
class temp
{
     private :
          static int count;         // Only one data item for all objects
     public :
          temp() { count++; }       // increment count when object created
          int getcount() { return count; }        // return count
};
int main()
{
     temp t1, t2, t3;        // create three objects
     cout << "\nCount is    " << t1.getcount( ); // each object
     cout << "\nCount is    " << t2.getcount( ); // sees the same
     cout << "\nCount is    " << t3.getcount( ); // value of count
     return 0;
}
```

Out put of the above program is as follows: (if it's still static)

Count is 3

Count is 3

Count is 3

Out put of the above program (if it's automatic)

Count is 1

Count is 1

Count is 1

13. Static Member Functions

Like static member variable, we can also have static member functions. A member function that is declared static has the following properties.

- A static functions can have access to only other static members (functions or variables) declared in the same class.
- A static member function cab be called using the class name (instead of its objects) as follows:

```
class-name :: function-name;
```

```
// Program to demonstrate static member function
#include <iostream>
using namespace std;
class test
{
     int code ;
     static int count;   // static member variable
     public:
          void setcode() { code = ++count; }
          void showcode() { cout << "Object number :" << code << endl; }
          static void showcount()                    // static member function
          {
                cout << "Count  :" << count << endl;
          }
};
int test :: count;
int main()
{
             test t1, t2;
             t1.setcode();
             t2.setcode();
             test::showcount();  // accessing static function
             test t3;
             t3.setcode();
             test::showcount();
             t1.showcode();
             t2.showcode();
             t3.showcode();
             return 0;
}
```

14. Friend Functions

```
class abc
{
   ....
   ....
public:
   ....
   ....
   friend void xyz(void);  // declaration
};
```

Private members cannot be accessed from outside the class. That is, a non-member function can't have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. It's simply achieved through **Friend** functions.

A friend function possesses certain special characteristics:

- It is not in the scope of the class to 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 the class. It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

```
// Program to demonstrate friend function
#include <iostream>
using namespace std;
class test
{
      int a;
      int b;
      public:
            void setvalue() { a = 25; b = 40; }
            friend float sum(test s);              // FRIEND declared
};
float sum(test s)
{
     return float (s.a + s.b ) / 2.0;         // s.a & s.b are the private members
                                              // of class test but they were accessible
                                              // by friend function
}
int main()
{
      test x;
      x.setvalue();
      cout << "Mean value  = " << sum(x) << endl;
      return 0;
}
```

One more example to implement a **friend functions as a bridge between two classes**.
The following program creates two objects of two classes and a function friendly to two classes.
In this example friend function is capable of accessing both classes data members and calculates the biggest of both class data members.

```
#include <iostream>
using namespace std;
class second;
class first
{
      int a;
      public:
         first(int temp) { a = temp; }
         friend void max(first, second);
};
class second
{
      int b;
      public:
          second(int temp) { b = temp; }
          friend void max(first, second);
};
void max(first f, second s)
{
      if ( f.a > s.b )                    // both first, second data members can be
           cout << "Max "<<  f.a;         // accessed thru friend max function
      else
           cout << "Max "<< s.b;
}
int main()
{
      first f(20);
      second s(30);
      max(f, s);
      return 0;
}
```

Ref: Object-oriented Programming in Turbo C++: Robert Lafore

previous | up | next
Learning C/C++ Step-By-Step - Page 12 | Learning C/C++ Step-By-Step - Page 14

add comment | view as pdf | print: this | all page(s)

Related Tutorials

- Beginner's Guide To c++
- An Explanation of Pointers (C++)