TATA ELXSI

OBJECT ORIENTED PROGRAMMING USING C++
Module 2

Learning & Development Team

Implementing Classes and Objects

Objectives

In this section, you will learn to:

- Understand C++ program
- Variables and data types
- Implement an object based on a class
- Describe the access specifiers Private, Public, & Protected
- Describe the scope resolution operator
- Describe the **this** pointer
- Constant Member functions
- Function Overloading

Understanding C++ Program :

```
/*
  This is a simple C++ program.
  File banner is to be written here…
*/

#include <iostream>            //Include the header file
using namespace std;

// A C++ program begins at main().
int main()
{
        cout << “Hello world“<<endl;
        return 0;
}
```

Data Types

- Primitive Data types: int, char, float and double.
- Arrays
  - Array size cannot be exactly equal to length of string.

    ex. char str[5]=“Hello” not allowed but char str[6]=“Hello” is allowed

- Pointers
  - Constant Pointer
    - char * const constptr = “Hello”;

  - pointer to a constant
    - int const *ptrconst=&a;

User-Defined Data types

➢ **Structures and Classes**

- Structures remain same as C
- Classes are similar to Structures with subtle differences

➢ **Enumerated Data Types**

- an **enum** is a set of integer constants
- compiler's default value assignment starts with 0, and each subsequent enumerator increments by 1

- enum { RED, BLUE, GREEN };

- can also be explicitly assigned value in declaration (which doesn't necessarily have to be unique)

- enum { RED=100, BLUE, GREEN=147 };

Variables

- symbols that represent values in a program
- have datatype and name
  - type-specifier identifier [= initial-value];
  - char cMyChar;
  - unsigned long nObjectID;
  - int iHours, iMinutes, iSeconds;
  - int iAnswer = 42;
  - float fMyTemp = 98.6;
- variable must be declared before it can be used

The Dot . Operator [Period]

- When you access a member of a class through a reference, you use the **dot operator**.
- The arrow operator is reserved for use with pointers only.
- Almost similar to Structures of C.

Implementing a Class and its Object

- Let us now represent a point on a two-dimensional plane as a user-defined data type.
- A point on a two-dimensional plane is represented by its x-coordinate and y-coordinate.
- The most basic operation on this Point data type would be to store valid screen coordinates into the data in the type.
- There may be need for operations which need to retrieve the x and y coordinates of a particular point object.

Example: A struct and its Object

For example,

```
struct point
 {
        int x_coord;
        int y_coord;
        void setx( int x)
        { x_coord = x; }

        void sety (int y)
        { y_coord = y; }

        int getx( void)
        {  return x_coord; }
        int gety( void)
        { return y_coord;}

}; // end of struct
```

Example: A Class and its Object

```
int main( )
{
   int a, b;  // a structure variable p1 of point type, struct
          // keyword not required
   point p1;
   p1.setx(22); // set the value of x_coord  of p1
   p1.sety(44); // set the value of y_coord of p1
   a = p1.getx( ); // return the value of the x_coord member of p1
   b = p1.gety( ); // return the value of the y_coord member of p1
}
```

Example: A Class and its Object

For example,

```
class point
 {
        int x_coord;
        int y_coord;
        void setx( int x)
        { x_coord = x; }

        void sety (int y)
        { y_coord = y; }

        int getx( void)
        {  return x_coord; }
        int gety( void)
        { return y_coord; }

};  // end of class
```

Accessibility of Struct Members

- Variables and methods declared within a struct are freely accessible to functions outside the structure declaration.

- **Therefore, all members in a structure are by default public.**

Accessibility of Class Members

- On the other hand, when a class declaration is used for the Point data type as depicted earlier, the data members and the member functions are accessible from only within the class.

- Data and methods within the class declaration will no longer be visible to functions outside the class point. The member functions to get and set the x and y coordinates can no longer be called from main( )

- **Therefore, all members in a class are by default private, thereby not being accessible outside the class.**

Access Specifiers

- The **private** access specifier is generally used to encapsulate or hide the member data in the class.

- The **public** access specifier is used to expose the member functions to the outside world, that is, to outside functions as interfaces to the class.

- The modified code for the class point is presented in the following slides:

Class Declaration for Point

```
class point
 {
        private:    int x_coord;    int y_coord;
        public:
        void setx( int x)
        { x_coord = x; }

        void sety (int y)
        { y_coord = y; }

        int getx( void)     {  return x_coord; }
        int gety( void)     {  return y_coord;}
```

Class Declaration for Point

```
main( )
{
  int a, b;
  // an object p1 of class type point, class keyword not required
  point p1;
  p1.setx(20); // set the value of x_coord  of object p1
  p1.sety(40); // set the value of y_coord of object p1
  a = p1.getx( ); // return the value of the x_coord member of object p1
  b = p1.gety( ); // return the value of the y_coord member of p1
}
```

The this pointer

Consider the following code:

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

class Simple
{
private:
    int id;

public:
     void setID(int id) { this->id = id; }
     int getID() { return this->id; }
};
```

The this pointer

```
int main()
{
    Simple simple;
    simple.setID(2);
    cout << simple.getID() << '\n';

   getchar();
    return 0;
}
```

The this pointer

- Each class member function contains an implicit pointer of its class type, named **this.**

- The **this** pointer, created automatically by the compiler, contains the address of the object through which the function is invoked.

- Therefore, when the member function setid( ) is invoked through simple, the function setid( ) implicitly receives the address of the object simple (***this**), and therefore, the id of simple is set.

Scope Resolution Operator ::

- we were defining all member functions within the body of the class.

- C++ provides the scope resolution operator **::** that allows the body of the member functions to be separated from the body of the class.

- Using the **:: operator**, the programmer can define a member function outside the class definition, without the function losing its connection to the class.

Scope Resolution Operator :: (Other Features)

- To access the global Variables .
- To define the static variables.
- To invoke the static functions.

And of course,

- To define the function outside the class.

Scope Resolution Operator ::

- Consider the following example:

```
class point
 {
   private:
    int x_coord;
    int y_coord;
   public:
    point (int x, int y);
    void setx (int x);
 };
```

```
point::point (int x, int y)
 {
   x_coord = x;
    y_coord = y;
 }
void point::setx( int x)
{ x_coord = x; }
```

Static Class Members – Static Data Members

- Both function and data members of a class can be made **static**.
- When you precede a member variable's declaration with the keyword static, you are telling the compiler that only one copy of that variable will exist.
- All objects of that class will share that variable.

Static Data Members

```
class static_demo
{
 private:
  static int data;          → Class Variable
  int    a,  b;             → Instance Variable
 public:
  void setValue ( int i, int j)
  {a = i; b = j; }

  void showValues( );
};
```

```
int static_demo::data = 20; // define the static variable
void static_demo::showValues( )
{
  cout << “this is static a:  “ << data;
  cout << this is non-static b:  “ <<a<< b; << ‘\n’;
}
```

Static Data Members

```
int main( )
 {
   static_demo x, y;
   x.set(1, 1); //set a to 1
   x.showValues( );
   y.set(2, 2); // change a to 2
   y.showValues ( );
   x.showValues ( );
/* Here, a has been changed for both x and y because a is
  shared by both objects */
   return 0;
  }
```

Static Data Members – Uses

- An interesting use of a static member variable is to keep track of the number of objects of a particular class type that is in existence. Consider the following example:

```
class counter_test
 {
   public:
    static int count;
    counter_test ( ) { count++; }
    ~counter_test ( ) { count--;}
  };
```

```
int counter_test::count;
void f( );
int main( )
 {
   counter_test ob1;
   cout << objects in existence:  " << counter_test::count << "\n";

   counter_test ob2;
   cout << objects in existence:  " << counter_test::count << "\n";
   f( );
   cout << objects in existence:  " << counter_test::count << "\n";
   return 0; }

void f( )
 {
    counter temp;
    cout << objects in existence:  " << counter_test::count << "\n";
    // temp is destroyed when f( ) returns
  }
```

A mutable Object Member

- A **const member function** cannot modify the state of its object.

- However, auxiliary data members (flags, reference counters) sometimes have to be modified by a const member function. Such data members can be declared mutable.

- A mutable member is never const, even if its object is const; therefore, it can be modified by a const member function.

- The following example demonstrates the use of this feature:

Eg. Program for Constant Member functions

```
class CMF {
    int value;
public:
    CMF(int v = 0) {value = v;}

    // We get compiler error if we add a line like "value = 100;"
    // in this function.
    int getValue() const{  return value;  }
};

int main() {
    CMF t(20);
    cout<<t.getValue();
    return 0;
}
```

Eg. Program for Constant Member functions with mutable

```
class CMF {
    mutable int value;
public:
    CMF(int v = 0) {value = v;}

    // We don't get compiler error if we add a line like "value = 100;"
    // in this function.
    int getValue() const{  return value;  }
};

int main() {
    CMF t(20);
    cout<<t.getValue();
    return 0;
}
```

Function Overloading

- Function overloading is the process of using the same name for two or more functions
- Each redefinition of the function must use either have different types of parameters or a different number of parameters.
- Two functions differing only in their return types cannot be overloaded.

Function Overloading Eg.

```
double myfunc(double d){
   return d;
}

int myfunc(int i){
   return i;
}

int main(){
   cout << myfunc(10) << myfunc(5.4);
   return 0;
}
```

Function Overloading Eg.

```
int myfunc(int i){
   return i;
}

int myfunc(int i, int j){
   return i*j;
}

int main(){
   cout << myfunc(10) << myfunc(4, 5);
   return 0;
}
```

Are you ready to solve…

1. ________ variables are not stored in objects.

   a. const b. volatile c. static d. all of them

   Ans: **c. static**

2. _________ specifier emphasizes on hiding the member data in the class.

   a. public b. private c. protected d. a&c.

   Ans: **b. private**

End of Module - 2

Disclaimer

- Some examples and concepts have been sourced from the below links and are open source material
    - **http://cppreference.com**
    - **www.cplusplus.com**

- References:
    - *C++: The Complete Reference* - 4th Edition by Herbert Schildt, Tata McGraw-Hill publications.
    - *The C++ Programming Language*- by Bjarne Stroustrup.
    - *Practical C++ Programming*- by Steve Oualline, O'Reilly publications.

Learning & Development Team

ITPB Road Whitefield

Bangalore 560 048 India

Tel +91 80 2297 9123

Fax +91 80 2841 1474

e-mail info@tataelxsi.com

www.tataelxsi.com

TATA ELXSI