Object-Oriented Programming Concepts

Feza BUZLUCA
Istanbul Technical University
Computer Engineering Department
http://faculty.itu.edu.tr/buzluca
http://www.buzluca.info

Overview

- Object-Oriented Approach
- Classes, Objects, Member Functions, and Data Members
- C++ Terminology
- Defining Methods As Inline Functions
- Defining Dynamic Objects
- Defining Arrays of Objects
- Controlling Access to Members
- Class and Struct in C++
- Friend Functions and Friend Classes
- `this` Pointer

Object-Oriented Approach

- When you approach a programming problem in an object-oriented language, you will try to divide the problem into objects
- Thinking in terms of objects, rather than functions, has a helpful effect on how easily you can design programs, because
  - real world consists of objects
  - there is a close match between objects in the programming sense and objects in the real world

Objects

- Many real-world objects have
  - attributes (characteristics that can change) and
  - abilities/responsibilities (things they can do)

Real-world object = Attributes (State) + Abilities (behavior, responsibility)
Programming object = Data + Functions

- Match between programming objects and real-world objects is result of combining data and member functions
- How can we define an object in a C++ program?

Classes

- Class
  - is a new data type which is used to define objects
  - serves as a plan or template
  - specifies what data and what functions will be included in objects of that class
  - is a description of similar objects
- Object
  - is an instance of a class

Defining a class does not create any objects

Example: Point Class

Example: A model (class) to define points in a graphics program

- Points on a plane must have two properties (states)
  - x- and y-coordinates: We can use two integer variables to represent these properties
- In our program, points should have following abilities (responsibilities)
  - Moving on plane: move function
  - Displaying their coordinates on screen: print function
  - Answering if they are at the origin (0,0) or not: isAtOrigin function

Example: Point Class Definition

```
class Point {          // declaration of Point class
  int x, y;            // attributes: x- and y-coordinates
 public:               // we will discuss it later
  void move(int, int);// function to move points
  void print();        // to print coordinates on screen
  bool isAtOrigin();   // is the point at the origin?
};                     // end of class declaration (Do not forget ";")
```

Attributes

Behavior, responsibilities

- In our example, first data and then function prototypes are written
  - It is also possible to write them in reverse order
- Data and functions in a class are called members of the class
- In our example, only the prototypes of the member functions are written in the class definition
- Function bodies may appear in other parts (in other files) of the program
- If the body of a function is written in the class definition, then this function is defined as an inline function (macro)

Example: Bodies of Member Functions

```
// ***** Bodies of Member Functions *****

// Move point
void Point::move(int newX, int newY)
{
   x = newX;                  // assigns new value to x-coordinate
   y = newY;                  // assigns new value to y-coordinate
}

// Print coordinates on the screen
void Point::print()
{
   cout << "X = " << x << ", Y = " << y << endl;
}

// Check if point is at the origin
bool Point::isAtOrigin()
{
   return (x == 0) && (y == 0); // if x = 0 AND y = 0, return true
}
```

Example: Creating Objects in Main

- Now, we have a model (template) to define point objects
  - We can create necessary points (objects) using the model

```
int main()
{
   Point   point1, point2;    // two objects defined: point1, point2
   point1.move(100, 50);      // point1 moves to (100, 50)
   point1.print();            // point1's coordinates to the screen
   point1.move(20, 65);       // point1 moves to (20, 65)
   point1.print();            // point1's coordinates to the screen
   if ( point1.isAtOrigin() ) // is point1 at (0, 0)?
      cout << "point1 is now at the origin (0, 0)" << endl;
   else
      cout << "point1 is NOT at the origin (0, 0)" << endl;
   point2.move(0, 0);         // point2 moves to (0, 0)
   if ( point2.isAtOrigin() ) // is point2 at (0, 0)?
      cout << "point2 is now at the origin (0, 0)" << endl;
   else
      cout << "point2 is NOT at the origin (0, 0)" << endl;
   return 0;
}
```

See Example e31.cpp

C++ Terminology

- Class
  - is a grouping of data and functions
  - is very much like an ANSI C struct (only a pattern/template to be used to create a variable that can be manipulated in a program)
  - is designed to provide certain services
- Object
  - is an instance of a class (similar to a variable defined as an instance of a type)
  - is what you actually use in a program
- An attribute is a data member of a class that can take different values for different instances (objects) of this class
  - Examples: Name of a student, coordinates of a point

C++ Terminology

- A method (member function) is a function contained within the class
  - Functions used within a class often referred to as methods in programming literature
  - Classes provide their services (carry out their responsibilities) with the help of their methods
- A message is the same thing as a function call
  - In object-oriented programming, we send messages instead of calling functions
  - For the time being, you can think of them as identical
  - Later, we will see that they are, in fact, slightly different
  - Messages are sent to objects to get some services from them

Conclusions So Far

- We have explored some features of object-oriented programming and C++
- Our programs consist of objects as the real world does
- Classes are living (active) data types used to define objects
- We can send messages (orders) to objects to tell them to perform a task
- Classes include both data and functions that act on this data (encapsulation)
- Consequently
  - Software objects are similar to real-world objects
  - Programs are easy to read and understand
  - It is easy to find errors
  - This approach supports modularity and teamwork

Defining Methods as Inline Functions (Macros)

- In the previous example (Example 3.1)
  - Only function prototypes of member functions were written in class definition
  - Bodies of methods were defined outside the class
- It is also possible to write bodies of methods in the class
  - Such methods are defined as inline functions

Defining Methods as Inline Functions: Example

- We can define isAtOrigin method of Point class as an inline function

```
class Point {          // definition of Point class
  int x, y;            // properties: x and y coordinates
 public:
  void move(int, int);// move point
  void print();        // print coordinates on the screen
  bool isAtOrigin()    // check if point at origin (inline function)
  {
       return (x == 0) && (y == 0);    // body of isAtOrigin
  }
};
```

Do not write long methods in the class declaration
It decreases the readability and performance of the program

Defining Dynamic Objects

- Classes can be used to define variables like built-in data types (int, float, char, etc.) of the compiler
  - For example, it is possible to define pointers to objects
- In the example below, two pointers (ptr1 and ptr2) to objects of type Point are defined

```
int main()
{
   Point *ptr1 = new Point; // allocate memory for obj. ptr1 points to
   Point *ptr2 = new Point; // allocate memory for obj. ptr2 points to
   ptr1->move(50, 50);      // 'move' message to obj. ptr1 points to
   ptr1->print();           // 'print' message to obj. ptr1 points to
   ptr2->move(100, 150);    // 'move' message to obj. ptr2 points to
   if( ptr2->isAtOrigin() ) // is object ptr2 points to at origin?
      cout << " Object ptr2 points to is at the origin." << endl;
   else
      cout << " Object ptr2 points to is NOT at the origin." << endl;
   delete ptr1;             // release the memory
   delete ptr2;
   return 0;
}
```


Defining Arrays of Objects

- We can define static and dynamic arrays of objects
- Example: a static array with ten elements of type Point

```
int main()
{
   Point   array[10]; // defining an array with ten objects
   array[0].move(15, 40);    // 'move' msg. to first element (index 0)
   array[1].move(75, 35);    // 'move' msg. to second element (index 1)
       :                     // message to other elements
   for (int i = 0; i < 10; i++) // 'print' message to all obj. in array
      array[i].print();
   return 0;
}
```

- We will see later how to define dynamic arrays of objects

Hidden Implementation

- We can divide programmers into two groups
  - Class creators (those who create new data types)
  - Client programmers (class consumers who use the data types in their applications)
- Goal of the class creator: to build a class that includes all necessary properties and abilities
  - Class should expose only what is necessary to the client programmer and keeps everything else hidden
- Goal of the client programmer: to collect a toolbox full of classes to use for rapid application development

Why Control Access to Members?

1. To keep client programmers' hands off portions they should not touch
   - Hidden parts are only necessary for the internal working of the data type, but not part of the interface that users need in order to solve their particular problems
2. If it is hidden, the client programmer cannot use it, which means that the class creator can change the hidden portion at will without worrying about the impact to anyone else
   - This protection also prevents accidental changes of states of objects

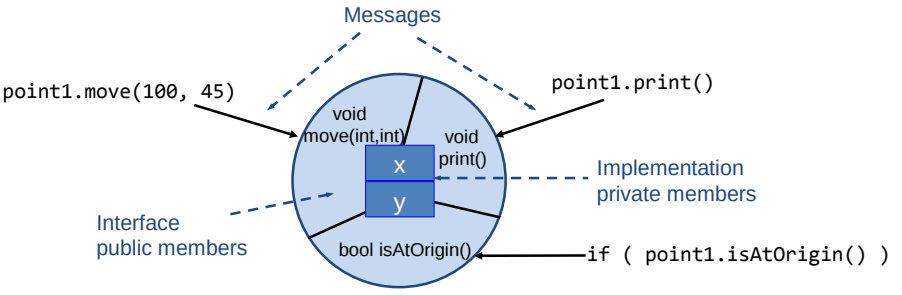
Access Specifiers: Public and Private

- Keywords public: , private: (and protected: as we will see later) are access specifiers used to control access to data members and functions of a class
- Private class members are only accessible to members of that class
- Public members may be accessed by any function in the program

Accessing Class Members

- Default access for classes is private
  - After each access specifier, the access that was invoked by that access specifier applies until the next access specifier or until the end of class declaration
- Primary purpose of public members is to present to the clients of the class a view of the services the class provides
  - This set of services forms the public interface of the class
- Private members are not accessible to the clients of a class
  - They form the implementation of the class

Accessing Class Members: Example



Accessing Class Members: Example

- Example: We modify the move function of the class Point
  - Clients of this class cannot move a point outside a window of 500 x 300

```
class Point {           // Point class
   int x, y;            // private members: x- and y-coordinates
  public:               // public members
   bool move(int, int); // move the point
   void print();        // print coordinates on screen
   bool isAtOrigin();   // check if point is at origin (0,0)
};
// function that moves the points ([0, 500] x [0, 300])
bool Point::move(int newX, int newY)
{
   if ( newX > 0 &&  newX < 500 &&    // if newX is in 0-500
        newY > 0 &&  newY < 300    ) // if newY is in 0-300
   {
      x = newX;       // assigns new value to x-coordinate
      y = newY;       // assigns new value to y-coordinate
      return true;    // input values are accepted
   }
   return false;      // input values are not accepted
}
```

Accessing Class Members: Example

- New move function returns a Boolean value to inform client programmer if input values are accepted

```
int main()
{
   Point p1; // define p1 object
   int x, y; // define two vars. to read some values from keyboard
   cout << "Input x- and y-coordinates ";
   cin >> x >> y;      // read two values from the keyboard
   if ( p1.move(x, y) )// send move message and check the result
      p1.print();      // if result OK, print coordinates on screen
   else
      cout << endl << "Input values are not accepted";
}
```

- It is not possible to assign a value to x or y directly outside the class

```
p1.x = -10;                    // ERROR! x is private
```


Class and Struct in C++

- class and struct keywords have very similar meanings in C++
  - They are both used to build object models
- Only difference is their default access
  - Private for class
  - Public for struct

Friend Functions and Friend Classes

- A function or entire class may be declared to be friend of another class
- A friend of a class has right to access all members (private, protected, and public) of class

```
class A {
  friend class B;      // Class B is a friend of class A
  private:             // private members of A
     int i;
     float f;
  public:              // public members of A
     void func1();     // not important
};

class B {              // Class B
    int j;
  public:
    void func2(A &s) { cout << s.i; } // B can access private
                                      // members of A
};
```

```
int main()
{
   A objA;
   B objB;
   objB.func2(objA);
   return 0;
}
```

A is not a friend of B
A cannot access private members of B

Friend Functions

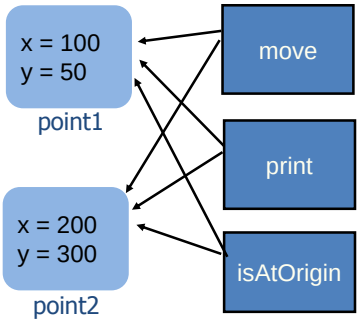
- A friend function has the right to access all members (private, protected, and public) of the class

```
class Point {                    // Point class
   friend void zeroOut(Point &);// friend function of Point
   int x, y;                     // private members: x- and y-coord.
  public:                        // public members
   bool move(int, int);          // move the point
   void print();                 // print coordinates on screen
   bool isAtOrigin();            // check if point is at origin (0,0)
};

// assign zero to all coordinates
void zeroOut(Point &p)       // not a member of any class
{
   p.x = 0;                  // assign zero to x of p
   p.y = 0;                  // assign zero to y of p
}
```

this Pointer

- Each object has its own data space in memory of computer
- When an object is defined, memory is allocated only for its data members
- Code of member functions created only once
- Each object of the same class uses the same function code



- How does C++ ensure that the proper object is referenced?
- C++ compiler maintains a pointer, called the this pointer

this Pointer

- A C++ compiler defines an object pointer this
- When a member function is called, this pointer contains the address of the object, for which the function is invoked
- So, member functions can access the data members using the pointer this

this Pointer: Example

- The compiler compiles our Point methods as follows:

```
void Point::move(int newX, int newY)  // function that moves points
{
   this->x = newX;            // assign new value to x-coordinate
   this->y = newY;            // assign new value to y-coordinate
}

void Point::print()           // print coordinates on screen
{
   cout << "X = " << this->x << ", Y = " << this->y << endl;
}
```

```
point1.move(50, 100);
```

```
this = &point1;   // address of object point1 is assigned to this
move(50, 100);    // and the method move is called
```


this Pointer: Another Example

- Programmers also can use this pointer in their programs
- Example: We add a new function to Point class: isFartherAway
  - To return address of object that is more distant to (0,0)

```
Point *Point::isFartherAway(Point &p)
{
   unsigned long x1 = x * x;             // x1 = x²
   unsigned long y1 = y * y;             // y1 = y²
   unsigned long x2 = p.x * p.x;
   unsigned long y2 = p.y * p.y;
   if ( (x1+y1) > (x2+y2) ) return this;// object returns its address
      else return &p;                    // address of incoming object
}
int main()
{
  Point point1, point2;              // two objects: point1, point2
  point1.move(100, 50);
  point2.move(20, 65);
  Point *ptr;                        // ptr is a pointer to points
  ptr = point1.isFartherAway(point2);
```

See Example e32.cpp

this Pointer: Example

- this pointer can also be used in methods if a parameter of the method has same name as one of the members of the class

```
class Point {              // Point class
   int x, y;               // private members: x- and y-coordinates
  public:                  // public members
   bool move(int, int);    // function that moves points
        :                  // other methods are omitted
};
// function that moves points ([0, 500] x [0, 300])
bool Point::move(int x, int y)  // parameters have the same names
{                               // as data members x and y
   if ( x > 0 &&  x < 500 &&    // if given x is in 0-500 and
        y > 0 &&  y < 300)      // if given y is in 0-300
   {
      this->x = x;              // assign given x value to member x
      this->y = y;              // assign given y value to member y
      return true;              // input values are accepted
   }
   return false;                // input values are not accepted
}
```