3 OO Programming Concepts

Content

- OOP Concepts
  - Class
    - Encapsulation
    - Information Hiding
  - Inheritance
  - Polymorphism

OOP Concepts

- When you approach a programming problem in an object-oriented language, you no longer ask how the problem will be divided into functions, but **how it will be divided into objects**.
- Thinking in terms of objects rather than functions has a helpful effect on how easily you can design programs. Because **the real world consists of objects** and there is a close match between objects in the programming sense and objects in the real world.

What is an Object?

- Many real-world objects have both **a state** (characteristics that can change) and **abilities** (things they can do).
- Real-world object=State (properties)+ Abilities (behavior)
- Programming objects = Data + Functions
- The match between programming objects and real-world objects is the result of combining data and member functions.
- How can we define an object in a C++ program?

Classes and Objects

- **Class** is a new data type which is used to define objects. A class serves as a plan, or a template. It specifies what data and what functions will be included in objects of that class. Writing a class doesn't create any objects.
- A class is a description of similar objects.
- **Object**s are instances of classes.

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 the following abilities (behavior):
  - Points can move on the plane: **move** function
  - Points can show their coordinates on the screen: **print** function
  - Points can answer the question whether they are on the zero point (0,0) or not: **is_zero** function

Class Definition: Point

```
class Point {              // Declaration of Point Class
   int x,y;                // Properties: x and y coordinates
 public:                   // We will discuss it later
   void move(int, int);    // A function to move the points
   void print();// to print the coordinates on the screen
   bool is_zero();         // is the point on the zero point(0,0)
};                         // End of class declaration (Don't forget ;)
```

In our example first data and then the function prototypes are written. It is also possible to write them in reverse order.

Data and functions in a class are called **member**s of the class.

In our example only the prototypes of the functions are written in the class declaration. The bodies may take place in other parts (in other files) of the program.

If the body of a function is written in the class declaration, then this function is defined as an inline function (macro).

Bodies of Member Functions


```
// A function to move the points
void Point::move(int new_x, int new_y) {
   x = new_x;              // assigns new value to x coordinate
   y = new_y;              // assigns new value to y coordinate
}
// To print the coordinates on the screen
void Point::print() {
   cout << "X= " << x << ", Y= " << y << endl;
}
// is the point on the zero point(0,0)
bool Point::is_zero() {
   return (x == 0) && (y == 0);          // if x=0 AND y=0 returns true
}
```

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

```
int main() {
   Point point1, point2;  // 2 object are defined: point1 and 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.is_zero() )  // is point1 on (0,0)?
      cout << "point1 is now on zero point(0,0)" << endl;
   else   cout << "point1 is NOT on zero point(0,0)" << endl;
   point2.move(0,0);      // point2 moves to (0,0)
   if( point2.is_zero() )  // is point2 on (0,0)?
      cout << "point2 is now on zero point(0,0)" << endl;
   else  cout << "point2 is NOT on zero point(0,0)" << endl;
   return 0;
}
```

C++ Terminology

- A **class** is a grouping of data and functions. A class is very much like a structure type as used in ANSI-C, it is only a pattern to be used to create a variable which can be manipulated in a program.
- An **object** is an instance of a class, which is similar to a variable defined as an instance of a type. An object is what you actually use in a program.
- A **method (member function)** is a function contained within the class. You will find the functions used within a class often referred to as methods in programming literature.
- 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.

Conclusion

- Until this slide we have discovered some features of the object-oriented programming and the C++.
- Our programs consist of object as the real world do.
- Classes are living (active) data types which are used to define objects. We can send messages (orders) to objects to enable them to do something.
- Classes include both data and the functions involved with these data (*encapsulation*). As the result:
- Software objects are similar to the real world objects,
- Programs are easy to read and understand,
- It is easy to find errors,
- It supports modularity and teamwork.

Defining Methods as inline Functions

- In the previous example (Example 3.1), only the prototypes of the member functions are written in the class declaration. The bodies of the methods are defined outside the class.
- It is also possible to write bodies of methods in the class. Such methods are defined as inline functions.
- For example the is_zero method of the Point class can be defined as an inline function as follows:

```
class Point{                    // Declaration of Point Class
   int x,y;                     // Properties: x and y coordinates
 public:
   void move(int, int);         // A function to move the points
   void print();        // to print the coordinates on the screen
   bool is_zero() {  // is the point on the zero point(0,0) inline function
       return (x == 0) && (y == 0);  // the body of is_zero
   }
};
```

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 to objects of type Point are defined.

```
int main()  {
    Point *ptr1 = new Point;  // allocating memory for the object pointed by ptr1
    Point *ptr2 = new Point;  // allocating memory for the object pointed by ptr2
    ptr1->move(50, 50);       // 'move' message to the object pointed by ptr1
    ptr1->print();            //'print' message to the object pointed by ptr1
    ptr2->move(100, 150);     // 'move' message to the object pointed by ptr2
    if( ptr2->is_zero() )     // is the object pointed by ptr2 on zero
        cout << " Object pointed by ptr2 is on zero." << endl;
    else cout << " Object pointed by ptr2 is NOT on zero." << endl;
    delete ptr1;                        // Releasing the memory
    delete ptr2;
    return 0;
}
```

Defining Array of Objects

- We may define static and dynamic arrays of objects. In the example below we see a static array with ten elements of type Point.
- We will see later how to define dynamic arrays of objects.

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

Controlling Access to Members

- We can divide programmers into two groups: class creators (those who create new data types) and client programmers (the class consumers who use the data types in their applications).
- The goal of the class creator is to build a class that includes all necessary properties and abilities. The class should expose only what's necessary to the client programmer and keeps everything else hidden.
- The goal of the client programmer is to collect a toolbox full of classes to use for rapid application development.
- The first reason for access control is to keep client programmers' hands off portions they shouldn't touch. The hidden parts are only necessary for the internal machinations of the data type but not part of the interface that users need in order to solve their particular problems.

Controlling Access to Members *Con't*

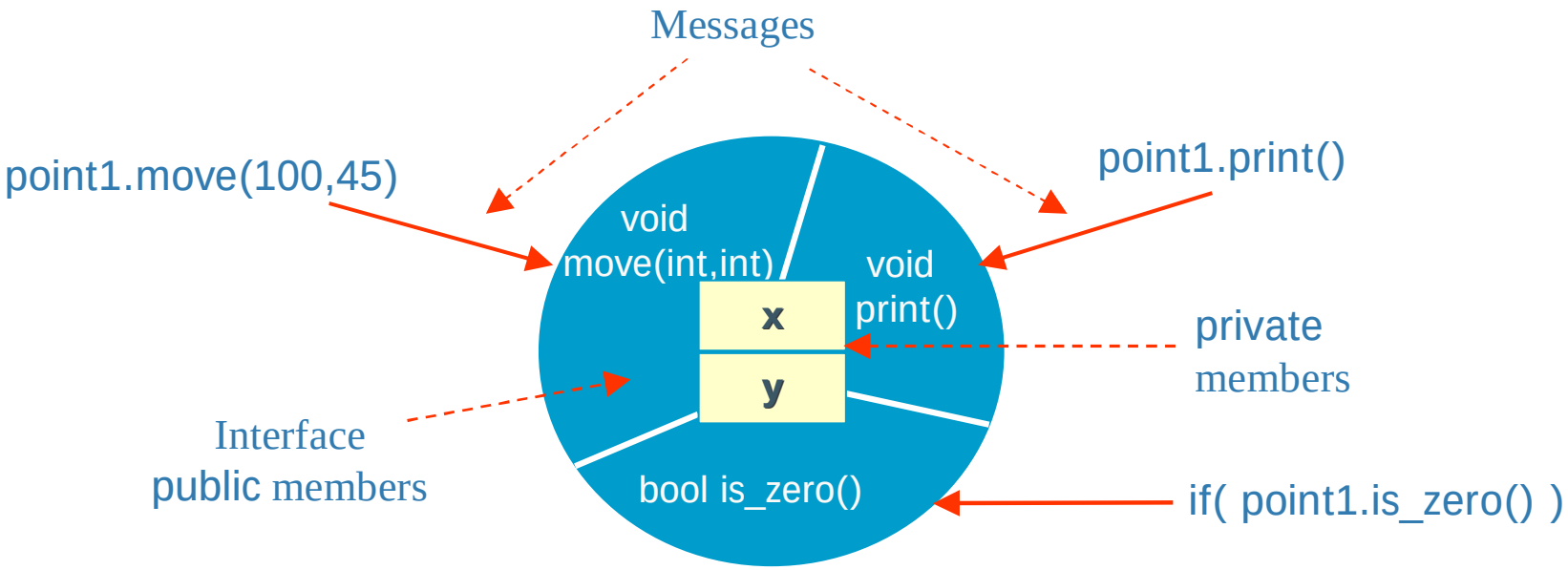
- The second reason for access control is that, if it's hidden, the client programmer can't use it, which means that the class creator can change the hidden portion at will without worrying about the impact to anyone else.
- This protection also prevents accidentally changes of states of objects.

Controlling Access to Members *Con't*

- The labels public: , private: (and protected: as we will see later) are used to control access to a class' data members and functions.
- Private class members can be accessed only by members of that class.
- Public members may be accessed by any function in the program.
- The default access mode for classes is private: After each label, the mode that was invoked by that label applies until the next label or until the end of class declaration.

Controlling Access to Members *Con't*

- The primary purpose of public members is to present to the class's clients a view of the services the class provides. This set of services forms the public interface of the class.
- The private members are not accessible to the clients of a class. They form the implementation of the class.



- Example: We modify the move function of the class Point. Clients of this class can not move a point outside a window with a size of 500x300.

```
class Point{                    // Point Class
    int x,y;                    // private members: x and y coordinates
  public:                                   // public members
    bool move(int, int);    // A function to move the points
    void print();           // to print the coordinates on the screen
    bool is_zero();         // is the point on the zero point(0,0)
};
// A function to move the points (0,500 x 0,300)
bool Point::move(int new_x, int new_y) {
   if( new_x > 0 &&  new_x < 500 &&              // if new_x is in 0-500
       new_y > 0 &&  new_y < 300) {      // if new_y is in 0-300
       x = new_x;                                // assigns new value to x coordinate
       y = new_y;                                // assigns new value to y coordinate
       return true;                              // input values are accepted
   }
   return false;                                 // input values are not accepted
}
```

- The new move function returns a boolean value to inform the client programmer whether the input values are accepted or not.
- Here is the main function:

```
int main()  {
     Point p1;        // p1 object is defined
     int x,y;         // Two variables to read some values from the keyboard
     cout << " Give 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 is OK print coordinates on the screen
     else  cout << "\nInput 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*

struct Keyword in C++

- ***class*** and ***struct*** keywords have very similar meaning in the C++.
- They both are used to build object models.
- The only difference is their default access mode.
- The default access mode for class is ***private***
- The default access mode for struct is ***public***

Friend Functions and Friend Classes

- A function or an entire class may be declared to be a friend of another class.
- A friend of a class has the right to access all members (private, protected or public) of the 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 fonk1(char *c);
};
class B{                                 // Class B
    int j;
  public:
    void fonk2(A &s) { cout << s.i; } // B can access private members of A
};
```

In this example, A is not a friend of B. A can not access private members of B.

Friend Functions and Friend Classes *Con't*

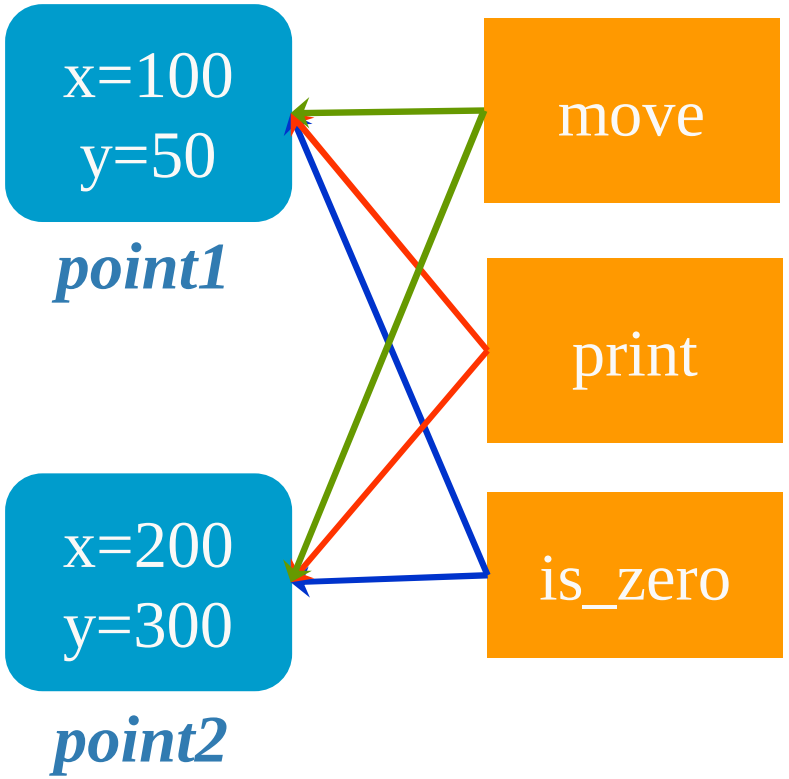
- A friend function has the right to access all members (private, protected or public) of the class.

```
class Point{                          // Point Class
    friend void zero(Point &);        // A friend function of Point
     int x,y;                         // private members: x and y coordinates
  public:                             // public members
     bool move(int, int);             // A function to move the points
     void print();                    // to print the coordinates on the screen
     bool is_zero();                  // is the point on the zero point(0,0)
};

// Assigns zero to all coordinates
void zero(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 the memory of the computer. When an object is defined, memory is allocated only for its data members.
- The code of member functions are 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.

- 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.
- Programmers also can use this pointer in their programs.
- **Example:** We add a new function to Point class: far_away.

This function will return the address of the object that has the largest distance from (0,0).

```
Point *Point::far_away(Point &p) {
   unsigned long x1 = x*x;                          // x1 = x^2
   unsigned long y1 = y*y;                          // y1 = y^2
   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;                          // The address of the incoming object
}
```

- this pointer can also be used in the methods if a parameter of the method has the 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);        // A function to move the points
          :                     // other methods are omitted
};
// A function to move the points (0,500 x 0,300)
bool Point::move(int x, int y)      // paramters has the same name as
{                                   // data members x and y
        if( x > 0 &&  x < 500 &&          // if given x is in 0-500
            y > 0 &&  y < 300) {    // if given y is in 0-300
            this->x = x;            // assigns given x value to member x
            this->y = y;            // assigns given y value to memeber y
            return true;            // input values are accepted
        }
        return false;               // input values are not accepted
}
```