2 OO Programming Concepts

Content

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

OOP Concepts

- When we approach a programming problem in an object-oriented language, we 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 we 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)+ Identity
- Programming objects = Data + Functions+ Identity
- 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.

- **Objects** 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 of the program.
- If the body of a function is written in the class declaration, then this function is defined as an inline function.

Bodies of Member Functions

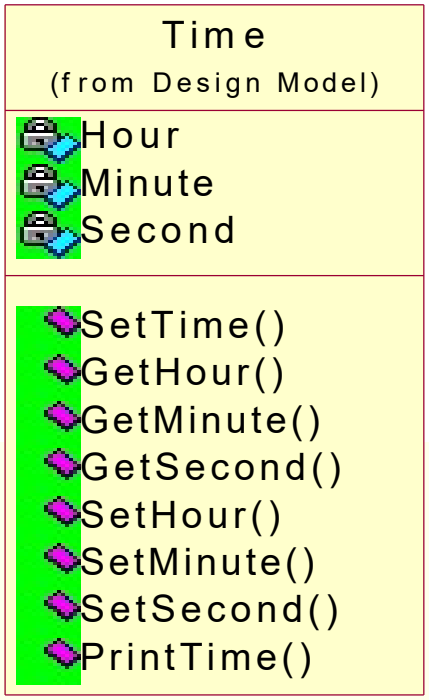

```
// A function to move the points
void Point::move(int new_x, int new_y) {
  x = new_x;          // assigns new value to x coordinate
  y = new_y;          // assigns new value to y coordinate
}
// To print the coordinates on the screen
void Point::print() {
  cout << "X= " << x << ", Y= " << y <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

```
class Time
   { int hour;
   int minute;
   int second ;
 public:
// Get Functions
   int    GetHour(){return hour;} ;
   int    GetMinute(){return minute;} ;
   int    GetSecond (){return second;} ;
// Set Functions
   void SetTime(int h,int m,int s){hour=h;minute=m;second=s;};
   void SetHour(int h){hour= (h>=0 && h<24) ? h : 0;} ;
   void SetMinute(int m){minute= (m>=0 && m<60) ? m : 0;} ;
   void SetSecond(int s){second= (s>=0 && s<60) ? s : 0;} ;
   void PrintTime();
};
```

UML Class Diagram



C++ Terminology

- A **class** is a grouping of data and functions. A class 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 we actually use in a program.
- A **method (member function)** is a function contained within the class. We 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, we 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.