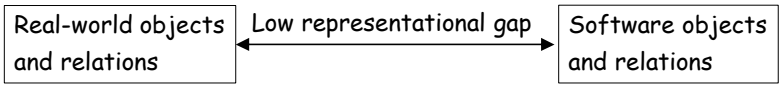
Object Oriented Programming Concepts

Remember: "The Object-Oriented Approach," slides 1.19 - 1.29.

Main approach:

The real world (problem) consists of objects.

The software system (solution) also consists of objects.



The close match between objects in the programming sense and objects in the real world increases the quality (understandability, readability) of the design.

To solve a problem in an object-oriented language, the programmer should consider three factors:

1. What are the **objects** that make up the problem domain?
2. What are the **responsibilities** of objects?
3. What are the **relations** between objects?

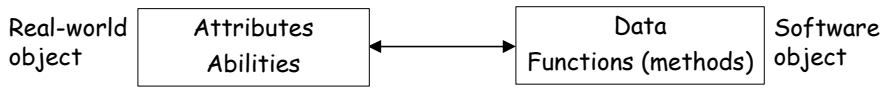
What is an object?

Real-world objects have two parts:

1. **Attributes** (*property* or *state*: characteristics that can change),
2. **Abilities** (or *behaviors*: things they can do or *responsibilities*).

Software objects (classes) also have two parts like real-world objects:

1. ***Data*** *represent attributes*,
2. ***Functions (methods)*** represent behavior.



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

Classes and Objects

Class is a user-defined data type that is used to define objects.

- A class serves as a plan or a template.
- It specifies what data and functions will be included in objects of that class.
- Writing a class does not create any objects.
- A class is a description of similar objects.

Objects are instances (variables) of classes.

Class declaration in C++:

```
class ClassName
{
public:
// Members (data and functions) that are accessible from outside the class
...
private:
// Members (data and functions) that are not accessible from outside the class
...
};
```


Example: A model (class) to define 2D points in a graphics program.

Based on the requirements of the stakeholders, points should have the following attributes and abilities (responsibilities):

Data: Attributes (states)

• **x** and **y** coordinates. We can use two integer variables to represent these attributes.

Functions: Abilities (responsibilities)

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

Declaration of the `Point` class

```
class Point {            // Declaration of the Point Class
public:                  // Open part                          Behavior,
  void move(int, int); // A function to move the points      responsibilities
  void print();         // Print the coordinates on the screen
  bool isOnZero();      // Is the point on the zero point(0,0)
private:                 // Data hiding                        Attributes
  int m_x{}, m_y{};     // Attribute: x and y coordinates
};                      // End of class declaration (Don't forget ;)
```

Example `Point` class (contd):

- Data and functions in a class are called **members** of the class.
- Convention: We add the prefix `m_` to the names of the member variables to easily distinguish them from function parameters and local variables.
- In our example, first, the `public` members and then the `private` members are written. It is also possible to write them in reverse order.
- We will discuss controlling access to members in the following subsection.
- Each of the member variables is initialized to `0`. You do not have to initialize member variables in this way.
- There are other ways of setting their values, as we will see in the next section (constructors).
- If member variables of fundamental types are not initialized by some mechanism, they will contain random values.
- In our example, only the prototypes of the functions are written in the class declaration.
- The bodies may take place in other parts (in different 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.

Example `Point` class (contd):

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

// A function to move the points
void Point::move(int new_x, int new_y)
{
  m_x = new_x;              // assigns a new value to the x coordinate
  m_y = new_y;              // assigns a new value to the y coordinate
}

// To print the coordinates on the screen
void Point::print()
{
  std::cout << "X= " << m_x << ", Y= " << m_y << std::endl;
}

// is the point on the zero point(0,0)
bool Point::isOnZero()
{
  return (m_x == 0) && (m_y == 0);     // if x=0 AND y=0 returns true
}
```

Defining objects of the `Point` class:

Now we have a type (model) 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
   point2.print();            // point2's coordinates to the screen
   point1.move(20,65);        // point1 moves to (20,65)
   if( point1.isOnZero() )           // is point1 on (0,0)?
      cout << "point1 is on zero point(0,0)" << endl;
   else
      cout << "point1 is NOT on zero point(0,0)" << endl;
   if( point2.isOnZero() )           // is point2 on (0,0)?
      cout << "point2 is on zero point(0,0)" << endl;
   else
      cout << "point2 is NOT on zero point(0,0)" << endl;
   return 0;
}
```

See Example e03_1.cpp

C++ TERMINOLOGY

- A **class** is a grouping of data and functions.

  A class is a type (a template, pattern, or model) used to create a variable that can be manipulated in a program.

  Classes are designed to give specific **services**.
- An **object** is an instance of a class, similar to a variable defined as an instance of a type. An object is what you 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.
  Example: Name of a student, coordinates of a point.
- 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.

  Classes fulfill their services (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.

Defining Methods as inline Functions (Macro)

In the previous example (Example e03_1.cpp), 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 `isOnZero` method of the `Point` class can be defined as an `inline` function as follows:

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

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.

Example: We define three pointers (`ptr1`, `ptr1`, and `ptr2`) to objects of type `Point`.

```
int main()
{
 Point *ptr1;           // Defining the pointer ptr1 to objects of the Point
 ptr1 = new Point;      // Allocating memory for the object pointed by ptr1
 Point *ptr2 = new Point;    // Pointer definition and memory allocation
 Point *ptr3 {new Point};    // Pointer definition and memory allocation
 ptr1->move(50, 50);        // 'move' message to the object pointed by ptr1
 ptr2->print();             //'print' message to the object pointed by ptr2
 if( ptr3->isOnZero() )     // is the object pointed to by ptr3 on zero
     cout << "The object pointed to by ptr2 is on zero." << endl;
  else
     cout << "The object pointed to by ptr2 is NOT on zero." << endl;
 delete ptr1;                                // Releasing memory
 delete ptr2;
 delete ptr3;
}
```

Defining Arrays of Objects

We may define static and dynamic arrays of objects.

The example below shows 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
  // 'move' message to the first element (indices 0)
  array[0].move(15, 40);     // point in[0] moves
  // 'move' message to the second element (indices 1)
  array[1].move(75, 35);     // point in[0] moves
      :                       // message to other elements
  // 'print' message to all objects in the array
  for (int i = 0; i < 10; i++){
       array[i].print();
       if (array[i].isOnZero())
           cout << "The point in " << i << " is on zero" << endl;
  }
 return 0;
}
```


Controlling Access to Members

We can divide programmers into two groups:

- *class creators:* Those who create new data types (declare classes)
- *client programmers (class users, object creators):* The class consumers who use the data types in their applications.

The goal (and responsibility) of the class creator is to build a class that includes all necessary properties and abilities.

The goal of the client programmer is to collect a toolbox full of classes to use for rapid application development.

The class creator is responsible for controlling access to data.

The class creator sets the rules, and class users must follow them.

Information hiding:

- The class should expose only what's needed to the client programmer (`public`) and
- keeps everything else **hidden** (`private`).

  The hidden parts are only necessary for the internal machinations of the data type but not part of the interface that users need to solve their particular problems.

Reasons for access control and its benefits:

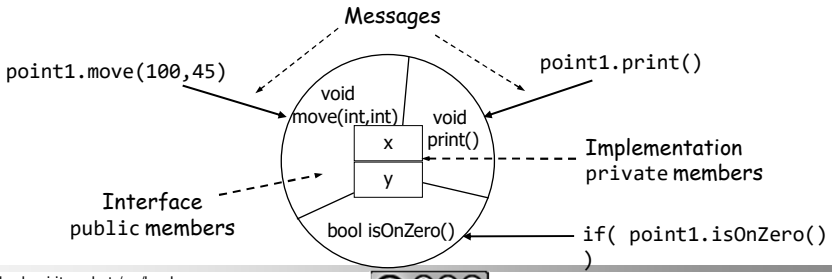
- To keep client programmers' hands off portions, they should not touch.
  A client programmer does not need to be aware of the internal private part of a class to use it.
  Learning only the public part (its interface) is sufficient.
- The client programmer cannot use the hidden part of a class.
  It means the class creator can change the hidden portion without worrying about its impact on anyone else.
- Information hiding also prevents accidental changes of attributes of objects.
- If attributes of an object get unexpected incorrect values, the usual suspects are member functions.
  This simplifies finding bugs.

Acces specifiers:

In C++, there are three access specifier labels:

`public:` , **`private:`** , and **`protected:`** (we will see it when we discuss inheritance).

- 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.
  Any function in the program may access public members.
- The **private** members are not accessible to the clients of a class. They form the ***implementation*** of the class.
  Private class members can be accessed only by members of that class.



Example: `Point` class with limits

According to stakeholder requirements, point objects may only move within a predetermined window. Thus, x and y coordinates may have limits.

For example, x must be between `0` and `500`, while y is between `0` and `300`.

Remember: The class creator is responsible for controlling access to data.

Clients of this class cannot move a point object outside a window with a size of `500x300`.

```
class Point{           // Declaration of the Point class with limits
public:
  bool move(int, int);      // A function to move points
  void print();             // to print coordinates on the screen
private:
 // Limits of x and y
 // Constants are usually defined as static members! (See static members)
  const int MIN_x{0};
  const int MAX_x{500};
  const int MIN_y{0};
  const int MAX_y{300};
  // x and y coordinates are initialized to their minimum values
  int m_x{MIN_x}, m_y{MIN_y};
};
```


Example: `Point` class with limits (contd)

The new `move` function returns a Boolean value to inform the client programmer whether or not the input values are accepted.

If the values fall within limits, they are accepted, the point moves, and the function returns `true`.

If the values are not within limits, the point does not move, and the function returns `false`.

```
bool Point::move(int new_x, int new_y)
{
  if (new_x >= MIN_x && new_x <= MAX_x &&   // if new_x is within limits
      new_y >= MIN_y && new_y <= MAX_y)     // if new_y is within limits
  {
    m_x = new_x;                  // assigns a new value to x coordinate
    m_y = new_y;                  // assigns a new value to y coordinate
    return true;                  // new values are not accepted
  }
  return false;                   // new values are not accepted
}
```

Example: `Point` class with limits (contd)

Here is the main function:

```
int main()
{
   Point point1;   // point1 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 (point1.move(x, y))     // send move message and check the result
     cout << "Input values are accepted" << endl;
   else
     cout << "Input values are NOT accepted" << endl;
   point1.print();    // Print coordinates on the screen
}
```

See Example e03_2.cpp

It is not possible to assign a value to `m_x` or `m_y` directly outside the class.

```
point1.m_x = -10;               //ERROR! m_x is private
```


Private methods (member functions):

Usually, data members are declared private, and methods are declared public.

However, methods may also be declared private if they are related solely to the internal mechanism of the class.

Private methods can only be called by other methods of the class.

Client programmers (object creators) cannot use private methods.

Example:

According to stakeholder requirements, the x and y coordinates of point objects must not exceed zero.

If a client of the class enters negative values to the `move` method, the point object resets its coordinates to zero.

Now, we will add a private `reset` method to the `Point` class that resets the coordinates to zero.

```
class Point{        // Declaration of the Point class with lower limits
public:
  :                   // public methods
private:
  void reset();       // private method
  :
};
```

See Example e03_3.cpp

Private methods (contd):

The move method checks the input values.

```
bool Point::move(int new_x, int new_y)
{
  // if the values are within the limits
  if (new_x >= MIN_x && new_y >= MIN_y)
  {
    m_x = new_x;              // assigns a new value to the x coordinate
    m_y = new_y;              // assigns a new value to the y coordinate
    return true;              // new values are accepted
  }
  reset();                    // calls reset
  return false;               // new values are not accepted
}
```

Client programmers (object creators) cannot call the `reset` method.

See Example e03_3.cpp

```
int main()
{
   Point point1;     // point1 object is defined
   point1.reset();   // ERROR! reset is private
   :
```


The order of public and private members:

You can alternate public and private sections as often as you want and put them in any order you wish.

Your class declarations become much easier to read and maintain if you consistently group related members together.

The default access mode for a class is **private**.

If you start with the private part, you do not even need to write the `private` label.

Example:

`private:` label is not necessary. It is the default mode in a class

```
class Point{                  // Declaration of the Point
  int m_x{}, m_y{};           // private part. x and y coordinates
public:
  bool move(int, int);        // A function to move points
  void print();               // to print coordinates on the screen
};
```

Our preference is, however, to write **the public part first**.

The order of public and private members (contd):

Grouping related members together:

```
class ClassName
{
 public:
 ...      // Group of related methods
 private:
 ...      // Related data members
 public:
 ...      // Group of methods
 private:
 ...      // Related data members
};
```

Convention:

- Put all public members first and all private members last.
  As a class user, you are normally primarily interested in its public interface and less so in its inner workings.
  You want to know what you can do with a class, not how it works.
  Therefore, we prefer to put the public interface first.
- We cluster related members and put variables after functions.

`struct` Keyword in C++:

`class` and **`struct`** keywords have very similar meanings in C++.

They both are used to build types.

The only difference is their default access mode.

The default access mode for a `class` is `private`.

The default access mode for the `struct` is `public`.

We usually use structures in C++ programs to define simple compound types that aggregate several variables.

Structures usually are simply encapsulating some publically accessible member variables (data).

Structures normally do not have many member functions.

You can, in principle, add private sections and member functions to a structure.

However, doing so is unconventional.

If your aim is not only aggregating data, then use a class.

Accessors and Mutators:

There will be situations where we want private member variables to be read or modified from outside the class.

For example, the user of the `Point` class may need to know the current values of the x and y coordinates.

Making these variables public is certainly not a good idea.

To allow private member variables to be read or modified from outside the class in *a controlled manner*, the creator of the class must provide special public methods.

Accessors (Getters):

Methods that retrieve (return) the values of member variables are referred to as *accessor* functions.

Convention: The accessor function for a data `member` is mostly called **`getMember()`**.

Because of this, these functions are more commonly referred to simply as *getters*.

Example: Accessors for the Point class with lower limits

```
public:
  int getX() { return m_x;}          // Accessor for x coordinate
  int getY() { return m_y;}          // Accessor for y coordinate
  int getMIN_x() { return MIN_x;} // Accessor for the limit of x
  int getMIN_y() { return MIN_y;} // Accessor for the limit of y
```


Mutators (Setters):

Methods that allow member variables to be modified are called **mutators**.

Convention: The accessor function for a data `member` is mostly called **`setMember()`**.

Because of this, these functions are more commonly referred to simply as **setters**.

Since we provide a member function to manipulate data rather than making the member variables public, we have the opportunity to perform integrity checks on the values given by the class users.

Example: Setters for the `Point` class with lower limits

```
class Point{
public:
  void setX(int new_x){
    if (new_x >= MIN_x) m_x = new_x;  // Accepts only valid values
  }
  void setY(int new_x){
    if (new_y >= MIN_y) m_y = new_y;  // Accepts only valid values
  }
 ...
}
```

See Example e03_4.cpp

The move method in our previous `Point` classes was a kind of mutator.

Remember: The class creator is responsible for controlling access to data. The class creator sets the rules, and class users must follow them.

Friend Functions and Friend Classes

Sometimes, it is useful to allow non-member functions to access non-public members of a class object.

The class creator may declare such a function to be a **friend** of the class.

A friend can access (to read and modify) any of the members of a class object, regardless of their access specification.

Example: Friend Function

A non-member `display` function is declared as a friend of the `Point` class. It can access private members of the `Point` class.

```
class Point{         // Declaration of the Point class
public:
 friend void display(Point&);   // non-member friend function
:
};

// Non-member function (outside of the Point class)
void display(Point &point){
  std::cout << "x= " << point.m_x << "y= " << point.m_y;
}
```

Call by reference

Private members are accessed directly.

```
int main()
{
  Point point1;
  point1.setX(10);
  point1.setY(20);
  display(point1);
```


Friend Class:

An entire class may also be declared to be a friend of another class.

All the methods of a *friend* class have unrestricted access to all the members of the class of which it has been declared a friend.

Example: Friend Class

A `GraphicTools` class is declared as a friend of the `Point` class.

```
class Point{                    // Declaration of the Point class
public:
 friend class GraphicTools;     // Friend class
:
};

class GraphicTools {
 public:
  void moveToZero(Point& point) {
    point.m_x = 0;
    point.m_y = 0;
  }
:
};
```

Another class (`GraphicTools`) can manipulate private members of the `Point` class directly.

```
int main()
{
  Point point1;
  point1.setX(10);
  point1.setY(20);
 // object of GraphicTools
  GraphicTools tool;
  tool.moveToZero(point1);
  :
}
```

point1 is on (0,0) now.

Friend Functions and Friend Classes (contd)

The friendship between classes is not a bidirectional relation.

Methods in the `GraphicTools` class can access all the members of the `Point` class, but methods in the Point class have no access to the private members of the `GraphicTools` class.

Friendship among classes is not transitive either; just because class `A` is a friend of class `B` and class `B` is a friend of class `C`, it doesn't follow that class `A` is a friend of class `C`.

Caution:

Friend declarations may undermine a fundamental principle of object-oriented programming: data hiding.

Therefore, they should only be used when absolutely necessary, and this situation does not occur frequently.

Use getters and setters, which provide safe access to class members.