Object-Oriented Programming Concepts

- Remember: "The Object-Oriented Approach," slides 1.9 - 1.16.
- 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

Software Classes and Objects

- A **Class** is a user (programmer)-defined data type that is used to define objects. It
  - serves as a plan or a template.
  - specifies what data and functions will be included in objects of that class.
  - is a description of similar objects.
  - Writing a class (class definition) does not create an object. It is simply a blueprint for an object.
- **Objects** are instances (variables) of classes.

Class declaration/definition 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) based on requirements
  - **x** and **y** coordinates. According to requirements, we can use two integer variables to represent these attributes.
- Functions: Abilities (responsibilities) based on requirements
  - Points can move on the plane: **move** function
  - Points can display their coordinates on the screen: **print** function
  - Points can answer the question of whether they are at the origin (0,0) or not: **isAtOrigin** function

Definition of the `Point` **class**

```
class Point {            // Declaration/definition of the Point Class
public:                  // Open part                                  Behavior,
  void move(int, int);   // A function to move the points              responsibilities
  void print() const;    // Print the coordinates on the screen
  bool isAtOrigin() const; // Is the point at the origin (0,0)
private:                 // Data hiding
  int m_x{}, m_y{};      // Attribute: x and y coordinates             Attributes
};                       // End of class declaration (Do not forget the semicolon ";")
```

Example: The `Point` class (cont'd):

- 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.
- Our example lists the `public` members first and then the `private` members (the reverse is also possible.
- We will discuss controlling access to members in the following subsection.
- Each member variable is initialized to `0` by using curly braces "`{}`" in its definition.
- There are other ways of setting their values, as we will see in the next section (constructors).
- If there is no mechanism initializing member variables of fundamental types, these variables will contain random values.
- In our example, only the prototypes (signatures, declarations) of the functions are written in the class definition.
- The bodies of the functions may appear in other parts (in different 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.

Example: The `Point` class (cont'd):

```
// ***** 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 to the screen
void Point::print() const
{
  std::println("X= {} , Y= {}", m_x, m_y); // {}s are replacement fields
}

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

Constant (`const`) methods do not modify the attributes of the class.
We will discuss the details in the upcoming slides.

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;              // Two objects 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.isAtOrigin() )          // is point1 at (0,0)?
      std::print("point1 is at the origin (0,0)");
   else
      std::print("point1 is NOT at the origin (0,0)");
   if( point2.isAtOrigin() )          // is point2 at (0,0)?
      std::print("point2 is at the origin (0,0)");
   else
      std::print("point2 is NOT at the origin (0,0)");
}
```

Class — Objects — Sending a message to an object

This program has some shortcomings as we have not yet covered all of the relevant information.
We will improve it later.

Example e03_1.cpp

We see the benefit of writing `std::` in this example.
It helps resolve the ambiguity between the `print` functions of `Point` and the Standard Library and explicitly specify which one we want to call.

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 provide 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.
  In object-oriented programming languages, the functions used within a class are often referred to as **methods**.
  Classes provide their services (or fulfill 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 being 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

- In the previous example (e03_1.cpp), only the prototypes of the member functions were written in the class declaration. The bodies of the 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.

  For example, the `isAtOrigin` method of the `Point` class can be defined as an `inline` function as follows:

```
class Point{                    // Definition of the Point Class
 public:
  // inline function
  bool isAtOrigin() const {
     return (m_x == 0) && (m_y == 0);  // the body is in the class
  }
  :                                    // Other methods of the class
 private:
   int m_x{}, m_y{};                   // x and y coordinates
};
```

- Remember: The compiler inserts the machine-language code of the inline function into the location of the function call.

- Do not write long methods in the class declaration. It degrades the readability and performance of the program.

Controlling Access to Members

- We can divide programmers into two groups:
  - *class creators:* Those who create new data types (define classes)
  - *client programmers (class users, object creators):* The class consumers who define objects and 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 (data) hiding:

- The class should
  - expose (make `public`) only what is needed by the client programmer and
  - keep everything else **hidden** (`private`).
- The hidden parts are only necessary for the internal working 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 the portions they should not touch.
  - A client programmer does not need to be aware of (understand or learn) 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.

  This means that the class creator can change the hidden portion without worrying about its impact on anyone else.
- Information hiding also prevents accidental changes to attributes of objects (increasing reliability and reducing the possibility of errors).
- If attributes of an object end up with unexpected, incorrect values, the usual suspects are member functions.

  This makes it easier to find the bugs.

Access specifiers:

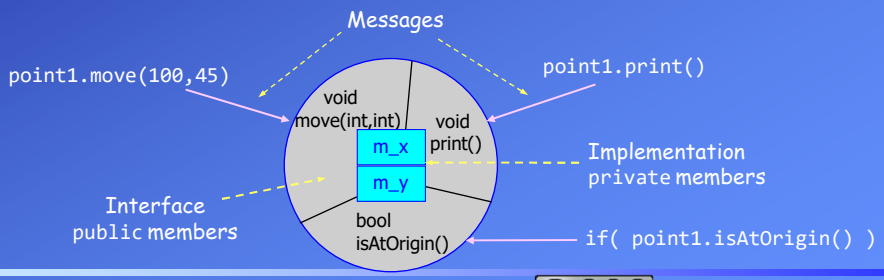
- In C++, there are three access specifier labels:

  `public:`, `private:`, and `protected:` (we will see it when we discuss inheritance).
- The primary purpose of **public** members is to present to the clients of the class 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.

  Only the members of a class can access its **private** members.



Example: The `Point` class with bounds

Requirement: According to stakeholder requirements, point objects may only move within a predetermined window (500 x 300).

Therefore, coordinates may have limits; 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 of a `500x300` window.

```
class Point{                    // Definition of the Point class with bounds
public:
  bool move(int, int);          // A function to move points
  void print() const;           // to print coordinates to the screen
private:
 // Limits of x and y
 // Constants are usually defined as static members! (See static members in Chapter 4)
  const int MIN_x{0};           // The lower bound for x is set to 0
  const int MAX_x{500};         // The upper bound for x is set to 500
  const int MIN_y{0};           // The lower bound for y is set to 0
  const int MAX_y{300};         // The upper bound for y is set to 300
  int m_x{MIN_x}, m_y{MIN_y}; // x and y coordinates are initialized to their minimum values
};
```


Example: The `Point` class with limits (cont'd)

- 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 the x coordinate
    m_y = new_y;                            // assigns a new value to the y coordinate
    return true;                            // new values are accepted
  }
  return false;                             // new values are not accepted
}
```

Example: The `Point` class with boundaries (limits) (cont'd)

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
   std::print("Enter the 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
     std::println("Input values are accepted");
   else
     std::println("Input values are NOT accepted");
   point1.print();                  // Print coordinates to the screen
}
```

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):

- Generallly, data members should be declared `private`, and methods should be declared `public`.
- 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:

Requirements:

- The x and y coordinates of point objects must not exceed zero.
- If a client of the class enters negative values as inputs to the `move` method, the point object resets its coordinates to zero.

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

```
class Point{          // Definition of the Point class with lower limits
public:
  :                   // public methods
private:
  void reset();       // private method (only the members of the class Point can call it)
  :
};
```

Example: Private methods (cont'd)

The move method calls the reset method if the input values are not accepted.

```
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 (new values are NOT accepted)
  return false;
}
```

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

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

Example e03_3.cpp

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{                // Definition 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() const;       // to print coordinates on the screen
};
```

- Our preference is to write **the public part first** because this is the part that interests the client programmer.

The order of `public` and `private` members (cont'd):

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++.
- 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.
- Usually, structures simply encapsulate publicly 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 aggregating data is not your only goal, 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 usually called **`getMember()`**.

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

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

```
public:
  int getX() const { return m_x;}          // Accessor for the x coordinate
  int getY() const { return m_y;}          // Accessor for the y coordinate
  int getMIN_x() const { return MIN_x;}    // Accessor for the limit of x
  int getMIN_y() const { 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 usually called **`setMember()`**.
- Because of this, these functions are commonly referred to as ***setters***.
- Since we provide a member function to manipulate data rather than make the member variables public, we have the opportunity to perform integrity checks on the values given by the class users.
- The `move` method in our previous `Point` classes was a kind of mutator.

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_y){
    if (new_y >= MIN_y) m_y = new_y;  // Accepts only valid values
  }
 ...
}
```

Example e03_4.cpp

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, allowing non-member functions to access non-public members of a class object is necessary.
- 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(const Point&);   // non-member friend function declaration
:
};

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

Call by reference

```
int main()
{
  Point point1;

  point1.setX(10);
  point1.setY(20);
  display(point1);
```

Not preferable! Private members are accessed directly.

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 declaration
:
};

class GraphicTools {
 public:
  void moveToOrigin(Point& point) {
    point.m_x = 0;     // private members of another class
    point.m_y = 0;
  }
:
};
```

Another class (`GraphicTools`) can manipulate private members of the `Point` class directly. **Not preferable!**

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

Friend Functions and Friend Classes (cont'd)

- The friendship between classes is not a bidirectional relation.
  For example, 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 (which is very rare).
- **Use getters and setters**, which provide safe access to class members.

Defining Dynamic Objects

- Using a class, we define a new data type that behaves exactly like the programming language's built-in data types (`int`, `float`, `char`, etc.).
  For example, it is possible to define pointers to objects.

Example: We define three pointers, `ptr1`, `ptr2`, and `ptr3`, to objects of type `Point`.

The Point object has not been created yet.

The object is created.

```
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->isAtOrigin() )        // is the object pointed to by ptr3 at the origin
     std::println("The object pointed to by ptr3 is at the origin.");
  else
     std::println("The object pointed to by ptr3 is NOT at the origin.");
  delete ptr1;                    // Releasing memory
  delete ptr2;
  delete ptr3;
}
```

Defining Arrays of Objects

- We can define static and dynamic arrays of objects.
- The example below shows a static array with ten elements of type `Point`.
- Later, we will see how to define dynamic arrays of objects.

```
int main()
{
  Point array[10];                  // defining an array with ten objects (points)
  // 'move' message to the first element (index 0)
  array[0].move(15, 40);            // point in[0] moves
  // 'move' message to the second element (index 1)
  array[1].move(75, 35);            // point in[1] 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].isAtOrigin())
           std::println("The point in {} is at the origin", i);
  }
  return 0;
}
```


Constant Objects and const Member Functions

Constant (const) objects:

- The programmer may use the keyword **const** to specify that an object is constant (not modifiable).
- Any attempt to modify the attributes of a constant (`const`) object directly or indirectly (by calling a function) results in a **compilation** error.
- Any member variable of a `const` object is itself a `const` variable and thus immutable.
- For example:

```
const Point fixedPoint;          // fixedPoint is a constant object
```

The object `fixedPoint` has the initial coordinates, and this point cannot be moved to another location.

```
fixedPoint.move(10,20);          // Error!
```

Constant (const) member functions:

- The programmer may define as **const** some member functions that do not modify any data members (attributes) of the object.
- Only `const` methods can operate on `const` objects.
- C++ compilers disallow non-constant method calls for `const` objects.

Constant (const) Member Functions (cont'd):

Example:

- We specify methods that do not modify an object's attributes as const.

```
class Point {
public:
  Point(int, int);                      // Constructor to initialize x and y coordinates
  double distanceFromOrigin() const;    // const method. The distance of a point from (0,0)
  void print() const;                   // const method prints coordinates to the screen
  // Getters are constant
  int getX() const { return m_x; }
  int getY() const { return m_y; }
  // Setters are not constant
  void setX(int);
  void setY(int);
  bool move(int, int);                  // A nonconstant method to move points
private:
  const int MIN_x{ 0 };                 // Lower bounds (minimums) are initialized to zero
  const int MIN_y{ 0 };
  int m_x{ MIN_x }, m_y{ MIN_y };       // x and y coordinates are initialized to minimums
};
```

A constant (const) method **cannot modify** the members of the class.
A const method containing code that modifies members cannot be compiled.

Constant Objects and const Member Functions (cont'd)

Example (cont'd):

```
// The constant method calculates and returns the distance of a point from (0,0)
double Point::distanceFromOrigin() const {
  return std::sqrt(m_x * m_x + m_y * m_y);    // distance from (0,0)
}

int main()
{
  const Point fixedPoint;                          // Constant object
  std::print("Distance from Origin = {}", fixedPoint.distanceFromOrigin());  // OK
  fixedPoint.print();                              // OK. Print the constant point
  fixedPoint.move(15, 25);                         // ERROR! fixedPoint is constant (cannot move)
  Point nonFixedPoint;                             // Nonconstant object
  nonFixedPoint.move(100, 200);                    // OK, nonconstant object can move
}
```

- A const method can invoke only other const methods because a const method is not allowed to alter an object's state either directly or indirectly (i.e., by invoking some non-const method).

Specify all member functions that do not change the object's attributes as const to avoid possible errors and to allow users of the class to define constant objects.

Constant Objects and const Member Functions (cont'd)

The mutable Keyword:

- Sometimes, we want to allow particular class members to be modifiable even for a const object.
- We can do this by specifying such attributes as **mutable**.

Example:

- We want to count how many times a point object is printed.
- We will add a mutable variable, m_printCount, to the Point class.

```
class Point {
public:
 Point(int, int);            // Constructor with two parameters to initialize x and y
 bool move(int, int);        // A nonconstant function to move points
 void print() const;         // A constant function to print
  :
private:
  :
 int m_x{ MIN_x }, m_y{ MIN_y };          // x and y coordinates are initialized
 mutable unsigned int m_printCount{};     // Mutable data member
};
```


The mutable Keyword (cont'd):

Example (cont'd):

```
// This method prints the coordinates to the screen and increments the m_printCount
void Point::print() const
{
  std::println("X= {} , Y= {}", m_x, m_y);
  std::println("Print count= {}", ++m_printCount);
}
```

A constant (const) method can modify mutable members.

Although the print method is specified as const, it can modify the mutable attribute printCount.

```
int main()
{
  const Point fixedPoint{ 10, 20 };   // Constant object
  fixedPoint.print();                 // m_printCount is incremented
  :
}
```

Example e03_5.cpp

Defining Classes in Modules (working with multiple files)

- In the previous examples, the declaration of the Point class, the bodies of its methods, and the main function were all written in the same file.
- However, in a real project with a large codebase, it is good practice to create separate files for related classes.
- The definition of the class can be written in a module interface, and the bodies of the methods can be defined in the module implementation.

Example:

Module interface file `shapes.ixx` for the `Point` class:

```
export module shape;          // module name can be different from the file name

export class Point {          // Declaration/Definition of the Point Class
 public:                      // Open part
   void move(int, int);       // A function to move the points
   void print() const;        // Print the coordinates to the screen
   bool isAtOrigin() const;   // Is the point at the origin (0,0)
 private:                     // Data hiding
   int m_x{}, m_y{};          // Attribute: x and y coordinates
};                            // End of class declaration
```


Defining Classes in Modules (cont'd)

Example (cont'd):

Module implementation file `shapes.cpp` for the `Point` class:

```
module shape;             // The name of the module (not file name)

import std;               // Standard module for println

void Point::move(int new_x, int new_y)
{
  m_x = new_x;
  m_y = new_y;
}
:  //----- Bodies of other methods ------
```

Example e03_6a.zip (`Point` class is in a module)

The `.cpp` file that contains the `main` function:

```
import shape;          // Importing the module

int main()
{
  ...
}
```

To avoid accidentally using the same name in conflicting situations, classes can be defined in namespaces.
Example: `namespace my_lib`

Example e03_6b.zip