ST. CHARLES COMMUNITY COLLEGE

CPT-182 - Programming in C++

Module 6

Classes and Objects

Dayu Wang

- **Objects**

  → A program is made up of items like variables and functions.

  To keep programs understandable, programmers often deal with higher-level groupings of those items known as objects.

  → In programming, an object is a grouping of data (variables) and operations that can be performed on that data (class-member functions).

  → [Example] Rectangle

  A rectangle has a width and a height.

  To represent each rectangle, it looks like we need to declare 2 variables, the `width` and `height`.

  However, it is more convenient to declare only 1 variable for each rectangle. But the variable has 2 "parts", the `width` part and `height` part.

  Also, a rectangle can calculate its area and perimeter based on its `width` and `height`.

  Therefore, we can group variable `width`, variable `height`, function `area()`, and function `perimeter()` together to form a single object, `rectangle`.

  → [Exercise] What data and functions can be grouped together to form a `Date` object?

  Variable `year`, `month`, and `day_of_month`, and function `print_date()`, and so on.

  → As long as it makes good sense, you can decide how you would like to group items.

- **Classes**

→ There are several viewpoints that we can use to understand classes in C++.

→ The first viewpoint to understand classes:

Classes are user-defined data types.

→ [Example] There is no ~~build-in data type~~ `Rectangle` in C++.

We can create our own `Rectangle` data type.

We can add data fields `width` and `height`, and class-member functions `area()` and `perimeter()`, inside the `Rectangle` class.

After the class is defined, we can create variables of `Rectangle` type in the `main()` function.

→ How to define a class in C++?

Convention 1

Do not ~~define two (or more) classes~~ in one file; only one class per file.

Convention 2

Filename is the same as the class name. For example, the filename of class `Rectangle` should be `Rectangle.h` and `Rectangle.cpp`.

Convention 3

You must separate the header file (`.h`) and implementation file (`.cpp`).

In other words, each class should be 2 files.

→ Some C++ code online does not ~~split a class into 2 files~~.

This is unprofessional.

In your assignments in this class, you will lose points if you failed to split a class into 2 files.

→ Class naming conventions

- In C++, the name of a user-defined class is first-letter capitalized.
- In C++, if the name of a user-defined class contains more than 1 word, then use underscore between words.

[Example] `Rectangle`, `Ordered_Vector`

→ C++ class is a generalization of the concept of `struct`.

`struct` is called compound data type, or record.

We will discuss the difference between `struct` and `class` later.

- **Basic Syntax of Class Definition**

```
1  class Rectangle {
2  public:
3      // Constructors
4      // Getters
5      // Setters
6      // Class-member functions
7  private:
8      // Data fields
9      // Class-member functions
10 };
```

Name of the class

Access variables separate the class into different sections.

There should be a semicolon after class definition.

→ **Access variables**

You will learn 3 different access variables in this class.

`private`

"`private`" means that all variables, `const` variables, and functions declared in this section are only accessible within the same class.

They are inaccessible outside the class.

`public`

"`public`" means that all variables, `const` variables, and functions declared in this section are accessible outside the class.

In other words, they are accessible "anywhere" in your source code.

You can define multiple `public` or `private` sections in a class, but it is not recommended.

By convention, `public` section comes before `private` section.

→ **What should be in `public` section? What should be in `private` section?**

Variables (data fields of the class) should be defined in `private` section.

[Example] A `Student` class may have data fields of `username` and `password`. Do you want your `password` be accessible everywhere outside the class?

Class-member functions (methods) should be defined in `public` section.

Users need to use those functions to manipulate the data in the class.

These are not always correct. But for beginners, this is "almost correct". So, you can use these as doctrines.

→ **If you do not put `public`/`private` in your class definition, then the default setting in C++ in `private`.**

This is bad practice.

→ **What is `struct`?**

`struct` behaves similar to `class`. However, in `struct`, there are no ~~`private`/`public` sections~~. Everything in a `struct` is `public`.

- **Header File and Implementation File**

  → In the class header file (.h), you only declare all the variables and functions.

  → Then, in the implementation file (.cpp), you implement all the functions declared in the header file.

  → [Example] The Rectangle class

Rectangle.h

```
1  #ifndef RECTANGLE_H
2  #define RECTANGLE_H
3  class Rectangle {
4  public:
5      // Class-member functions
6
7      // Returns the area of the rectangle.
8      unsigned int area() const;
9      // Returns the perimeter of the rectangle.
10     unsigned int perimeter() const;
11 private:
12     // Data fields
13     unsigned int width;
14     unsigned int height;
15 };
16 #endif
```


In the implementation file, Rectangle.cpp, you need to pay attention to the following 3 things:

1) You need to #include the header file.
2) In Microsoft Visual Studio, header files should be placed in the "Header Files" folder; implementation files should be placed in the "Source Files" folder.
3) To access a class-member function in the implementation file, you need to prepend "Rectangle::" before the function name (after the return type).

Rectangle.cpp

```
1  #include "Rectangle.h"
2
3  /** Calculates the area of the rectangle.
4      @returns: calculated area of the rectangle
5  */
6  unsigned int Rectangle::area() const { return width * height; }
7
8  /** Calculates the perimeter of the rectangle.
9      @returns: calculated perimeter of the rectangle
10 */
11 unsigned int Rectangle::perimeter() const {
12     return 2 * (width + height);
13 }
```

- **Getters and Setters**

→ **If `width` and `height` are `private`, how can we access (or modify) them in the `main()` function?**

We need to define functions to access (and modify) the `private` data fields in C++ classes.

→ **Getters**

Getters are special class-member functions that are used to access the `private` data fields.

1) **Getters do not ~~take any arguments~~.**
2) **Each getter only has a single return statement in it.**
3) **Getters are `const` functions, because they do not change the object itself (e.g., not changing `width`, nor `height`).**

In "Rectangle.h"

```
1  // Getters
2  unsigned int get_width() const;
3  unsigned int get_height() const;
```

In "Rectangle.cpp"

```
1  // Getters
2  unsigned int Rectangle::get_width() const { return width; }
3  unsigned int Rectangle::get_height() const { return height; }
```


→ **Setters**

Setters are special class-member functions that are used to modify the `private` data fields.

1) **Setters take arguments that represent the updated values.**
2) **Setters do not have a ~~return value~~ (return `void`).**
3) **Setters are not ~~`const` functions~~, because they do change the object.**

In "Rectangle.h"

```
1  // Setters
2  void set_width(unsigned int);
3  void set_height(unsigned int);
```

In "Rectangle.cpp"

```
1  // Setters
2  void Rectangle::set_width(unsigned int new_width) { width = new_width; }
3  void Rectangle::set_height(unsigned int new_height) {
4      height = new_height;
5  }
```

- **Now we can create `Rectangle` objects in the `main()` function.**

  → **In the file containing the `main()` function, you only need to `#include` the header file (`.h`), not ~~the implementation file (`.cpp`)~~.**

```
1  int main() {
2      Rectangle rect;  // Create a rectangle variable (object).
3      rect.set_width(10);  // Set the width to 10.
4      rect.set_height(20);  // Set the height to 20.
5      cout << "Area: " << rect.area() << endl;
6      system("pause");
7      return 0;
8  }
```

```
Console
Area: 200
```

- **What is the difference between class and object?**

  → **Class is more like a "template".**

  → **You can create multiple objects that belong to the same class.**

```
1  Rectangle rect_1, rect_2, rect_3;  // Create 3 rectangles.
```


- **Constructors**

  → **C++ does not ~~initialize variables~~ for the user.**

```
1  int main() {
2      Rectangle rect;
3      cout << "Width: " << rect.get_width() << endl;
4      cout << "Height: " << rect.get_height() << endl;
5      system("pause");
6      return 0;
7  }
```

```
Console
Width: 3435973836
Height: 3435973836
```

C++ users need to tell the compiler what are the initial values of all the data fields (variables) in the class.

→ **Constructors are special class-member functions that will be called automatically when an object of the class is being created.**

1) **Constructors themselves are class-member functions.**
2) **Constructors have the same name as the class name.**
3) **Constructors do not have ~~return type~~.**

[Pitfall] Constructors do not ~~return void~~. They have no ~~return type~~.

→ The constructor that takes no ~~argument~~ is called default constructor.

In "Rectangle.h"

```
1  Rectangle();  // Default constructor
```

In "Rectangle.cpp"

```
1  Rectangle::Rectangle() {  // Default constructor
2      width = 0;
3      height = 0;
4  }
```

Code can be more professional.

In "Rectangle.cpp"

```
1  Rectangle::Rectangle() : width(0), height(0) {}  // Professional
```

For any class, default constructor is a must.

→ Constructor taking initial values of width and height

In "Rectangle.h"

```
1  // Constructors
2  Rectangle();  // Default constructor
3  Rectangle(unsigned int, unsigned int);  // Overload the constructor.
```

In "Rectangle.cpp"

```
1  Rectangle::Rectangle(unsigned int initial_width, unsigned int
2  initial_height) : width(initial_width), height(initial_height) {}
```


→ The previous two constructors can be combined into one constructor.

In "Rectangle.h"

```
1  Rectangle(unsigned int = 0, unsigned int = 0);
```

We are using default parameter values here.

In "Rectangle.cpp"

```
1  Rectangle::Rectangle(unsigned int width, unsigned int height) :
2      width(width), height(height) {}
```

```
1  int main() {
2      Rectangle r1;  // Calls the default constructor.
3      Rectangle r2(10, 20);  // Calls the constructor with initial values.
4      cout << "R1: " << r1.get_width() << ' ' << r1.get_height() << endl;
5      cout << "R2: " << r2.get_width() << ' ' << r2.get_height() << endl;
6      system("pause");
7      return 0;
8  }
```

Console

```
R1: 0 0
R2: 10 20
```

- **Keyword const**

→ **Different positions of the const keyword have different meanings.**

→ **(1) const before data type in variable declarations**

```
1  const double PI = 3.1416;
```

Here, keyword const means that the variable is a constant variable whose value cannot be changed in subsequent code.

→ **(2) const before return type in function definitions**

```
1  /** Returns the element at user-specified index in a vector
2      @param vec: vector to search for the element
3      @param index: index of the element to return
4      @return: element at the user-specified index in the vector
5  */
6  const int& element(vector<int>& vec, size_t index) {
7      return vec.at(index);
8  }
```

Here, keyword const means that the return value of the function is a const reference to an integer.

"const reference" means that the integer cannot be ~~modified~~.

→ **(3) const before a reference type of a function parameter**

```
1  /** Returns the element at user-specified index in a vector
2      @param vec: vector to search for the element
3      @param index: index of the element to return
4      @return: element at the user-specified index in the vector
5  */
6  int element(const vector<int>& vec, size_t index) {
7      return vec.at(index);
8  }
```

Here, keyword const means that the parameter is passed by const reference.

→ [Exercise] Understanding reference and const reference

One of the following 4 function definitions is wrong. Find that one.

```
1  int& element(vector<int>& vec, size_t index) {
2      return vec.at(index);
3  }
```
A

```
1  const int& element(const vector<int>& vec, size_t index) {
2      return vec.at(index);
3  }
```
B

```
1  const int& element(vector<int>& vec, size_t index) {
2      return vec.at(index);
3  }
```
C

```
1  int& element(const vector<int>& vec, size_t index) {
2      return vec.at(index);
3  }
```
D

[Hint] If a vector is passed by const reference, then all the element in it are const references.

→ You can treat a reference as a const reference; you cannot treat a const reference as a reference.

You can treat your smartphone as a calculator; you cannot treat your calculator as a smartphone.

→ Understanding reference and value

What is the difference between the following two versions?

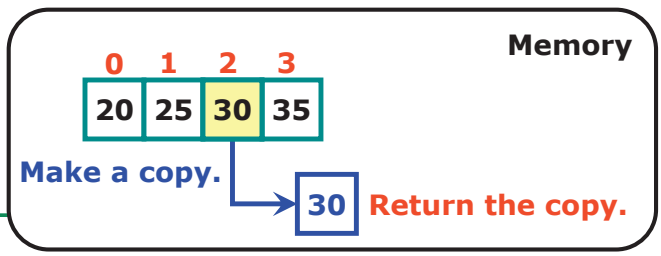
```
1  int element(vector<int>& vec, size_t index) {
2      return vec.at(index);
3  }
```
E

```
1  int& element(vector<int>& vec, size_t index) {
2      return vec.at(index);
3  }
```
F

E returns a value.

[Example] vec: {20, 25, 30, 35}

index: 2



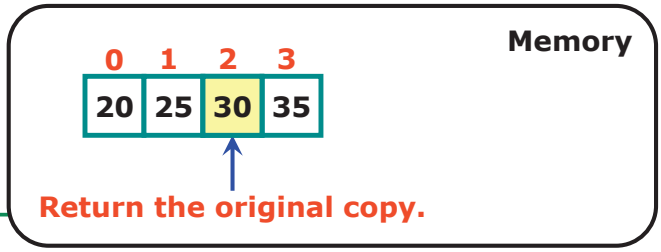

```
1  int main() {
2      vector<int> vec = { 20, 25, 30, 35 };
3      int value = element(vec, 2);
4      value = 100;  // Change the value.
5      for (int i = 0; i < vec.size(); i++) { cout << vec.at(i) << ' '; }
6      system("pause");
7      return 0;
8  }
```

```
Console  20 25 30 35
```

F returns a reference.

[Example] vec: {20, 25, 30, 35}

index: 2




```
1  int main() {
2      vector<int> vec = { 20, 25, 30, 35 };
3      int value = element(vec, 2);
4      value = 100;  // Change the value.
5      for (int i = 0; i < vec.size(); i++) { cout << vec.at(i) << ' '; }
6      system("pause");
7      return 0;
8  }
```

Console: 20 25 100 35

→ In an assignment statement, "A = B", A (left side) must be modifiable; B (right side) can be const reference (unmodifiable).

Therefore, a modifiable reference is called lvalue; const reference (unmodifiable) is called rvalue.

→ You need to be clear about value, reference, and const reference.

int, int&, and const int& are different data types.

→ (4) const at the end of a class-member function declaration

The function is called const function.

Here, keyword const means that the function does not ~~change the current object~~.

In other words, the function does not ~~change any data field~~ of the class.

→ [Important] In your assignments, if a class-member function does not ~~change the class data fields~~, then you must put const at the end of the function declaration to make it a const function.

In your assignments, for every class-member function, it is your responsibility to correctly determine:

1) Function's return type and return value
2) Number of function parameters
3) Data type of each function parameter
4) Each parameter should be passed by value, reference, or const reference.
5) Whether the function should be a const function.

Failing to do this correctly will result in losing points.

→ **[Exercise] Write a class-member function in the `Rectangle` class, `is_square()`, that returns `true` if the rectangle is a square, `false` otherwise.**

1) **Return type: `bool`**
2) **Number of function parameters: 0**
3) **Data type of function parameters: N/A**
4) **Each parameters passed by value, reference, or `const` reference: N/A**
5) **Is the function a `const` function? Yes**

In "Rectangle.h"

```
1  bool is_square() const;  // Tests whether the rectangle is a square.
```

In "Rectangle.cpp"

```
1  /** Tests whether the rectangle is a square.
2      @return: {true} if the rectangle is a square; {false} otherwise
3  */
4  bool Rectangle::is_square() const { return width == height; }
```


→ **[Exercise] Write a class-member function in the `Rectangle` class, `is_smaller_than()`, which takes a `Rectangle` object (another rectangle) as its only argument. The function returns `true` if the current rectangle has smaller area than the other rectangle, `false` otherwise.**

1) **Return type: `bool`**
2) **Number of function parameters: 1**
3) **Data type of the function parameter: `Rectangle`**
4) **The parameter passed by value, reference, or `const` reference? `const` reference**
5) **Is the function a `const` function? Yes**

In "Rectangle.h"

```
1  // Compares the area of this rectangle with another rectangle.
2  bool is_smaller_than(const Rectangle&) const;
```

In "Rectangle.cpp"

```
1  /** Compares the area of this rectangle with another rectangle.
2      @param other: the other rectangle to compare with this rectangle
3      @return: {true} if this rectangle has smaller area;
4               {false} otherwise
5  */
6  bool Rectangle::is_smaller_than(const Rectangle& other) const {
7      return area() < other.area();
8  }
```

- **Static variables and static class-member functions**

  → `width` and `height` belong to each object.

```
1  Rectangle rect_1, rect_2;
```

`rect_1` has its own `width` and `height`.

`rect_2` has its own `width` and `height`.

Changing `rect_1`'s `width` will not affect `rect_2`.

Changing `rect_2`'s `height` will not affect `rect_1`.

→ [Conclusion] `width` and `height` are variables of each object.

→ In a C++ class, a static variable is a variable of the class instead of ~~a variable of each class object~~.

Static variables are independent of any class object, and can be accessed without ~~creating a class object~~.

Changing a static variable will affect all the objects that belong to the class.

- **[Example] The `Date` Class**

In "Date.h"

```
1  class Date {
2  public:
3      // Static field
4      static string format;  // Either "US", "Euro", or "Asian"
5
6      // Constructor
7      Date(unsigned int = 0, unsigned int = 0, unsigned int = 0);
8
9      // Class-member function
10     void print_date(ostream&) const;
11         // Prints a date to an output stream.
12
13 private:
14     // Data fields
15     unsigned int year, month, day_of_month;
16 };
```

In "Date.cpp"

```cpp
// Static field
string Date::format = "US";

// Constructor
Date::Date(unsigned int year, unsigned int month, unsigned int
day_of_month) : year(year), month(month), day_of_month(day_of_month) {}

// Class-member function

/** Prints the date to an output stream.
    @param out: an output stream to show the date
*/
void Date::print_date(ostream& out) const {
    if (format == "US") {
        out << month << '/' << day_of_month << '/' << year << endl;
    }
    if (format == "Euro") {
        out << day_of_month << '/' << month << '/' << year << endl;
    }
    if (format == "Asian") {
        out << year << '-' << month << '-' << day_of_month << endl;
    }
}
```



```cpp
int main() {
    Date d_1(2022, 7, 2), d_2(2015, 5, 4);
    d_1.print_date(cout);
    d_2.print_date(cout);

    Date::format = "Euro";
    cout << endl;
    d_1.print_date(cout);
    d_2.print_date(cout);

    Date::format = "Asian";
    cout << endl;
    d_1.print_date(cout);
    d_2.print_date(cout);

    system("pause");
    return 0;
}
```

Console

```
7/2/2022
5/4/2015

2/7/2022
4/5/2015

2022-7-2
2015-5-4
```

→ **For static variables (or functions), you need to use `[class_name]::` to access them.**

For instance variables (or functions), you need to use `[object_name].` to access them.

→ **In C++ classes, a `const` variable does not mean ~~it is `static`~~.**

→ If a function is `static`, then it cannot be a ~~`const` function~~.

A `const function` indicates that the function cannot change the object.

However, in a `static function`, there is no ~~object~~ exists, since a `static` function directly belongs to the class.

Therefore, a `static` function cannot be a ~~`const` function~~.

→ A `static` function can only call `static variables` and other `static functions`.

They cannot call any ~~instance variables~~ or ~~instance functions~~.

An instance variable (or instance function) requires an object to be created.

However, in a `static function`, there is no ~~object~~ exists, since a `static` function directly belongs to the class.

Therefore, a `static` function can only call `static variables` and other `static functions`.

- **Friend Functions and Friend Classes**

→ In C++, a class may set another class, or a non-class-member function, as a friend class (friend function).

→ A class's `friend` class (`friend` function) can access the entire `private` section of the class.

[Example] The `Circle` class

In "Circle.h"

```
1  class Circle {
2  public:
3      friend class Rectangle;
4      // Constructor
5      Circle(double = 0);
6      // Getter
7      double get_radius() const;
8      // Setter
9      void set_radius(double);
10     // Class-member function
11     double area() const;
12     double circumference() const;
13 private:
14     static const double PI;
15     double radius;
16 };
```

`Circle` sets `Rectangle` as friend.

So `Rectangle` can access the entire `private` section of `Circle`.

`Circle` cannot access the `private` section of `Rectangle`.

In "Circle.cpp"

```cpp
// Static field
const double Circle::PI = std::atan(1) * 4;
// Constructor
Circle::Circle(double radius) : radius(radius) {}
// Getter
double Circle::get_radius() const { return radius; }
// Setter
void Circle::set_radius(double new_radius) { radius = new_radius; }
// Class-member function
double Circle::area() const { return PI * radius * radius; }
double Circle::circumference() const { return 2 * PI * radius; }
```

In "Rectangle.h"

```cpp
// Tests whether the rectangle can be placed inside a circle.
bool can_be_placed_inside(const Circle&) const;
```

In "Rectangle.cpp"

```cpp
bool Rectangle::can_be_placed_inside(const Circle& cir) const {
    // If the rectangle's diagonal length is smaller than the circle's
    // diameter, then it can be placed inside the circle.
    return sqrt(width * width + height * height) <= 2 * cir.radius;
}
```


[Example] The transpose() function

In "Rectangle.h"

```cpp
friend void transpose(Rectangle&);  // Transposes a rectangle.
```

Class Rectangle sets function transpose() as a friend, so function transpose(), even if it is a non-class-member function, can access the entire private section of the Rectangle class.

In "Rectangle.cpp"

```cpp
/** Transposes a rectangle.
    @param rect: rectangle to transpose
*/
void transpose(Rectangle& rect) { swap(rect.width, rect.height); }
```

Why we should not use "Rectangle::transpose"?

"Rectangle::transpose" indicates that function transpose() is a class-member function of class Rectangle.

A friend function indicates that the function is a non-class-member function.

Therefore, we do not use "~~Rectangle::transpose~~".

→ **friend functions cannot be ~~const functions~~.**

→ **friend functions cannot be ~~static functions~~.**

A friend function neither ~~belongs to an object~~, nor ~~belongs to the class~~.

- **Three basic features of object-oriented programming**

  1) **Encapsulation (discussed)**
  2) **Inheritance (in future lectures)**
  3) **Polymorphism (in future lectures)**

- **A class encapsulates some data and some functions to manipulate the data in it.**

  → `friend` **classes and** `friend` **functions break the ~~encapsulation feature~~.**

  Only C++ supports `friend` **classes and** `friend` **functions.**