C++ - Module 02

Ad-hoc polymorphism, operator overloading and Orthodox Canonical class form

Summary:
This document contains the exercises of Module 02 from C++ modules.

Version: 7

Contents

Chapter I

Introduction

C++ is a general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language, or "C with Classes" (source: Wikipedia).

The goal of these modules is to introduce you to **Object-Oriented Programming**. This will be the starting point of your C++ journey. Many languages are recommended to learn OOP. We decided to choose C++ since it's derived from your old friend C. Because this is a complex language, and in order to keep things simple, your code will comply with the C++98 standard.

We are aware modern C++ is way different in a lot of aspects. So if you want to become a proficient C++ developer, it's up to you to go further after the 42 Common Core!

Chapter II

General rules

Compiling

- Compile your code with `c++` and the flags `-Wall -Wextra -Werror`
- Your code should still compile if you add the flag `-std=c++98`

Formatting and naming conventions

- The exercise directories will be named this way: `ex00`, `ex01`, ... , `exn`
- Name your files, classes, functions, member functions and attributes as required in the guidelines.
- Write class names in **UpperCamelCase** format. Files containing class code will always be named according to the class name. For instance: `ClassName.hpp`/`ClassName.h`, `ClassName.cpp`, or `ClassName.tpp`. Then, if you have a header file containing the definition of a class "BrickWall" standing for a brick wall, its name will be `BrickWall.hpp`.
- Unless specified otherwise, every output messages must be ended by a new-line character and displayed to the standard output.
- *Goodbye Norminette!* No coding style is enforced in the C++ modules. You can follow your favorite one. But keep in mind that a code your peer-evaluators can't understand is a code they can't grade. Do your best to write a clean and readable code.

Allowed/Forbidden

You are not coding in C anymore. Time to C++! Therefore:

- You are allowed to use almost everything from the standard library. Thus, instead of sticking to what you already know, it would be smart to use as much as possible the C++-ish versions of the C functions you are used to.
- However, you can't use any other external library. It means C++11 (and derived forms) and `Boost` libraries are forbidden. The following functions are forbidden too: `*printf()`, `*alloc()` and `free()`. If you use them, your grade will be 0 and that's it.

- Note that unless explicitly stated otherwise, the using namespace <ns_name> and friend keywords are forbidden. Otherwise, your grade will be -42.

- **You are allowed to use the STL in Module 08 only.** That means: no **Containers** (vector/list/map/and so forth) and no **Algorithms** (anything that requires to include the <algorithm> header) until then. Otherwise, your grade will be -42.

A few design requirements

- Memory leakage occurs in C++ too. When you allocate memory (by using the new keyword), you must avoid **memory leaks**.

- From Module 02 to Module 08, your classes must be designed in the **Orthodox Canonical Form, except when explicitely stated otherwise**.

- Any function implementation put in a header file (except for function templates) means 0 to the exercise.

- You should be able to use each of your headers independently from others. Thus, they must include all the dependencies they need. However, you must avoid the problem of double inclusion by adding **include guards**. Otherwise, your grade will be 0.

Read me

- You can add some additional files if you need to (i.e., to split your code). As these assignments are not verified by a program, feel free to do so as long as you turn in the mandatory files.

- Sometimes, the guidelines of an exercise look short but the examples can show requirements that are not explicitly written in the instructions.

- Read each module completely before starting! Really, do it.

- By Odin, by Thor! Use your brain!!!

> You will have to implement a lot of classes. This can seem tedious, unless you're able to script your favorite text editor.

> You are given a certain amount of freedom to complete the exercises. However, follow the mandatory rules and don't be lazy. You would miss a lot of useful information! Do not hesitate to read about theoretical concepts.

Chapter III

New rules

From now on, all your classes must be designed in the **Orthodox Canonical Form**, unless explicitely stated otherwise. Then, they will implement the four required member functions below:

- Default constructor
- Copy constructor
- Copy assignment operator
- Destructor

Split your class code into two files. The header file (.hpp/.h) contains the class definition whereas the source file (.cpp) contains the implementation.

Chapter IV

Exercise 00: My First Class in Orthodox Canonical Form

	Exercise : 00
My First Class in Orthodox Canonical Form	
Turn-in directory : *ex*00/	
Files to turn in : `Makefile, main.cpp, Fixed.{h, hpp}, Fixed.cpp`	
Forbidden functions : `None`	

You think you know integers and floating-point numbers. How cute.

Please read this 3 pages article (1, 2, 3) to discover that you don't. Go on, read it.

Until today, every number you used in your code were basically either integers or floating-point numbers, or any of their variants (`short`, `char`, `long`, `double`, and so forth). After reading the article above, it's safe to assume that integers and floating-point numbers have opposite caracteristics.

But today, things will change. You are going to discover a new and awesome number type: **fixed-point numbers**! Forever missing from the scalar types of most languages, fixed-point numbers offer a valuable balance between performance, accuracy, range and precision. That explains why fixed-point numbers are particularly applicable to computer graphics, sound processing or scientific programming, just to name a few.

As C++ lacks fixed-point numbers, you're going to add them. This article from Berkeley is a good start. If you have no idea what Berkeley University is, read this section of its wikipedia page.

Create a class in Orthodox Canonical Form that represents a fixed-point number:

- Private members:
    - An **integer** to store the fixed-point number value.
    - A **static constant integer** to store the number of fractional bits. Its value will always be the integer literal `8`.
- Public members:
    - A default constructor that initializes the fixed-point number value to `0`.
    - A copy constructor.
    - A copy assignment operator overload.
    - A destructor.
    - A member function `int getRawBits( void ) const;` that returns the raw value of the fixed-point value.
    - A member function `void setRawBits( int const raw );` that sets the raw value of the fixed-point number.

Running this code:

```
#include <iostream>

int         main( void ) {

  Fixed a;
  Fixed b( a );
  Fixed c;

  c = b;

  std::cout << a.getRawBits() << std::endl;
  std::cout << b.getRawBits() << std::endl;
  std::cout << c.getRawBits() << std::endl;

  return 0;
}
```

Should output something similar to:

```
$> ./a.out
Default constructor called
Copy constructor called
Copy assignment operator called // <-- This line may be missing depending on your implementation
getRawBits member function called
Default constructor called
Copy assignment operator called
getRawBits member function called
getRawBits member function called
0
getRawBits member function called
0
getRawBits member function called
0
Destructor called
Destructor called
Destructor called
$>
```

Chapter V

Exercise 01: Towards a more useful fixed-point number class

Exercise 01
Towards a more useful fixed-point number class
Turn-in directory : *ex*01/
Files to turn in : `Makefile, main.cpp, Fixed.{h, hpp}, Fixed.cpp`
Allowed functions : `roundf (from <cmath>)`

The previous exercise was a good start but our class is pretty useless. It can only represent the value `0.0`.

Add the following public constructors and public member functions to your class:

- A constructor that takes a constant integer as a parameter.
  It converts it to the corresponding fixed-point value. The fractional bits value is initialized to 8 like in exercise 00.
- A constructor that takes a constant floating-point number as a parameter.
  It converts it to the corresponding fixed-point value. The fractional bits value is initialized to 8 like in exercise 00.
- A member function `float toFloat( void ) const;`
  that converts the fixed-point value to a floating-point value.
- A member function `int toInt( void ) const;`
  that converts the fixed-point value to an integer value.

And add the following function to the **Fixed** class files:

- An overload of the insertion («) operator that inserts a floating-point representation of the fixed-point number into the output stream object passed as parameter.

Running this code:

```
#include <iostream>

int    main( void ) {

    Fixed       a;
    Fixed const b( 10 );
    Fixed const c( 42.42f );
    Fixed const d( b );

    a = Fixed( 1234.4321f );

    std::cout << "a is " << a << std::endl;
    std::cout << "b is " << b << std::endl;
    std::cout << "c is " << c << std::endl;
    std::cout << "d is " << d << std::endl;

    std::cout << "a is " << a.toInt() << " as integer" << std::endl;
    std::cout << "b is " << b.toInt() << " as integer" << std::endl;
    std::cout << "c is " << c.toInt() << " as integer" << std::endl;
    std::cout << "d is " << d.toInt() << " as integer" << std::endl;

    return 0;
}
```

Should output something similar to:

```
$> ./a.out
Default constructor called
Int constructor called
Float constructor called
Copy constructor called
Copy assignment operator called
Float constructor called
Copy assignment operator called
Destructor called
a is 1234.43
b is 10
c is 42.4219
d is 10
a is 1234 as integer
b is 10 as integer
c is 42 as integer
d is 10 as integer
Destructor called
Destructor called
Destructor called
Destructor called
$>
```

Chapter VI

Exercise 02: Now we're talking

Exercise 02
Now we're talking
Turn-in directory : *ex*02/
Files to turn in : Makefile, main.cpp, Fixed.{h, hpp}, Fixed.cpp
Allowed functions : roundf (from <cmath>)

Add public member functions to your class to overload the following operators:

- The 6 comparison operators: >, <, >=, <=, == and !=.
- The 4 arithmetic operators: +, -, *, and /.
- The 4 increment/decrement (pre-increment and post-increment, pre-decrement and post-decrement) operators, that will increase or decrease the fixed-point value from the smallest representable ϵ such as $1 + \epsilon > 1$.

Add these four public overloaded member functions to your class:

- A static member function min that takes as parameters two references on fixed-point numbers, and returns a reference to the smallest one.
- A static member function min that takes as parameters two references to **constant** fixed-point numbers, and returns a reference to the smallest one.
- A static member function max that takes as parameters two references on fixed-point numbers, and returns a reference to the greatest one.
- A static member function max that takes as parameters two references to **constant** fixed-point numbers, and returns a reference to the greatest one.

It's up to you to test every feature of your class. However, running the code below:

```
#include <iostream>

int     main( void ) {

    Fixed           a;
    Fixed const     b( Fixed( 5.05f ) * Fixed( 2 ) );

    std::cout << a << std::endl;
    std::cout << ++a << std::endl;
    std::cout << a << std::endl;
    std::cout << a++ << std::endl;
    std::cout << a << std::endl;

    std::cout << b << std::endl;

    std::cout << Fixed::max( a, b ) << std::endl;

    return 0;
}
```

Should output something like (for greater readability, the constructor/destructor messages are removed in the example below):

```
$> ./a.out
0
0.00390625
0.00390625
0.00390625
0.0078125
10.1016
10.1016
$>
```

Chapter VII

Exercise 03: BSP

	Exercise 03
BSP	
Turn-in directory : *ex*03/	
Files to turn in : Makefile, main.cpp, Fixed.{h, hpp}, Fixed.cpp, Point.{h, hpp}, Point.cpp, bsp.cpp	
Allowed functions : roundf (from <cmath>)	

Now that you have a functional **Fixed** class, it would be nice to use it.

Implement a function which indicates whether a point is inside of a triangle or not. Very useful, isn't it?

BSP stands for Binary space partitioning. You are welcome. :)

You can pass this module without doing exercise 03.

Let's start by creating the class **Point** in Orthodox Canonical Form that represents a 2D point:

- Private members:
  - A Fixed const attribute `x`.
  - A Fixed const attribute `y`.
  - Anything else useful.
- Public members:
  - A default constructor that initializes `x` and `y` to `0`.
  - A constructor that takes as parameters two constant floating-point numbers. It initializes `x` and `y` with those parameters.
  - A copy constructor.
  - A copy assignment operator overload.
  - A destructor.
  - Anything else useful.

To conclude, implement the following function in the appropriate file:

```
bool bsp( Point const a, Point const b, Point const c, Point const point);
```

- a, b, c: The vertices of our beloved triangle.
- point: The point to check.
- Returns: True if the point is inside the triangle. False otherwise.
  Thus, if the point is a vertex or on edge, it will return False.

Implement and turn in your own tests to ensure that your class behaves as expected.