PIC 10B Discussion: Week 1 – Thurs

Alex Lin

Spring 2017
Book: Big C++ (2ed), by Cay S. Horstmann, and Timothy A. Budd
Chapter 5 Classes

Chapter Goals:

▶ To be able to implement your own classes
▶ To master the separation of interface and implementation.
▶ To understand the concept of encapsulation.
▶ To design and implement accessor and mutator member functions.
▶ To understand object construction.
▶ To learn how to distribute a program over multiple source files.
Section 5.1: Discovering Classes

What is a class?

A class is a representation of a concept. It allows one to manifest a concept into code by grouping variables/data and having functions that allow one to mutate those variables/data.
Section 5.1: Discovering Classes

An example of a class

```cpp
#include <iostream>
using namespace std;

class Rectangle {
   int width, height;
public:
   void set_values(int, int);
   int area() const {return width * height;}
};

void Rectangle::set_values(int x, int y)
{
   width = x;
   height = y;
}

int main()
{
   Rectangle rect;
   rect.set_values(3, 4);
   cout << "area: " << rect.area();
   return 0;
}

(Remark: Always put a semicolon after the closing brace of a class definition.)
Section 5.2: Interfaces
Private, Protected, Public

- Every class in C++ has three types of access modifiers: private, protected, and public. These are applied to variables and functions of your class. By default (i.e. if no access modifiers are specified) members of a class are private.
- A private member is only accessible within the class.
- A public member can be accessed anywhere outside the class, but within the program.
- A protected member is like a private member, except it can be accessed by child classes. This access modifier comes up when dealing with inheritance.
Section 5.2: Interfaces

Examples of the public and private access modifiers

Using our previous example:

```cpp
#include <iostream>
using namespace std;

class Rectangle {
private:
    int width, height;  // private members, only accessible within class
public:
    void set_values(int x, int y);  // public member
    int area() const {return width*height;}  // public member
};

void Rectangle::set_values(int x, int y)
{
    width = x;
    height = y;
}

int main ()
{
    Rectangle rect;
    rect.set_values(3,4);
    cout << "area: " << rect.area();
    return 0;
}
```

Usually the mutator functions are public, while the variables/data they mutate are private.
Section 5.2: Interfaces

Example of the protected access modifier

```cpp
#include <iostream>
using namespace std;

class Box {
    protected:
        double width;
};

class SmallBox:Box // SmallBox is the derived class. {
    public:
        void setSmallWidth( double wid );
        double getSmallWidth( void );
};

// Member functions of child class
double SmallBox::getSmallWidth(void) {
    return width ;
}

void SmallBox::setSmallWidth( double wid ) {
    width = wid;
}

int main( ) {
    SmallBox box;

    // set box width using member function
    box.setSmallWidth(5.0);
    cout << "Width of box : " << box.getSmallWidth() << endl;

    return 0;
}
```
Section 5.2: Interface

Constructor

- The purpose of a constructor is to initialize objects when they are created. A constructor without parameters is called a *default constructor*.
- Every class should have a default constructor.
Section 5.2: Interface
Functions: Accessor v.s. Mutator

- An accessor member function merely gives back data of an object, and does not modify the object. Accessor functions must be tagged with a `const`.
- A mutator member function changes the state of the object.
#include <iostream>
#include <string>

using namespace std;

class Product
{
public:
    Product(); // Constructs a product with score 0 and price 1.
    void read(); // Reads in this product object.

    /*
     * Compares two product objects.
     * @param b the object to compare with this object
     * @return true if this object is better than b
     */
    bool is_better_than(Product b) const;

    void print() const; // Prints this product object.

private:
    string name;
    double price;
    int score;
};

Product::Product()
{
    price = 1;
    score = 0;
}
Section 5.3: Encapsulation

- Each object of a class stores certain data that are set by the constructor, and which may change throughout the lifetime of the object by the mutator functions. This data is collectively called the state of the object.

- Every class has a private implementation: data fields that store the state of an object. Because these fields are private, only the constructor and member functions can access them.

- So these data fields are hidden from the programmer. They are part of the implementation details and are of no concern to the user of the class.

- The act of hiding implementation details is called encapsulation.
Section 5.3: Encapsulation

Benefits

An example from Section 5.3:

class Time
{
public:
    Time();
    Time(int hrs, int min, int sec);
    void add_seconds(long s);
    int get_seconds() const;
    int get_minutes() const;
    int get_hours() const;
    long seconds_from() const;
private:
    ... // Hidden data representation
};
...

Time liftoff(19, 30, 0);
...
// Looks like the liftoff is getting delayed by another six hours
// Wont compile, but suppose it did
liftoff.hours = liftoff.hours + 6;
Section 5.3: Encapsulation

Benefits

- Avoids unnecessary access that could potentially cause user-created bugs.
- In larger programs, it is typical that implementation details need to change over time, e.g. because you want to make your program more efficient or capable. As long as the users of the program do not depend on the implementation details, you are free to change them.
Section 5.4: Member Functions

Provide details for every member function that is advertised in your program.

class Product
{
    public:
        Product();
        void read();
        bool is_better_than(Product b) const;
        void print() const;
    private:
        string name;
        double price;
        int score;
};

...

void Product::read()
{
    cout << "Please enter the model name: ";
    getline(cin, name);
    cout << "Please enter the price: ";
    cin >> price;
    cout << "Please enter the score: ";
    cin >> score;
    string remainder; // Read remainder of line
    getline(cin, remainder);
}
const correctness

Declare all accessor functions in C++ with the const keyword. If you don't, you build classes that other programmer cannot reuse. An example:

class Product
{
    ...
    void print() const;
    ...
};
...
class Order
{
    public:
    ...
    void print() const;
    private:
    string customer;
    Product article;
    ...
};
void Order::print() const
{
    cout << customer << "\n";
    article.print(); // Error if Product::print not const
    ...
}

Refuses to compile because article is an object of class Product, and Product::print is not const, so the compiler suspects that the call article.print() may modify article. But article is a data field of Order, and Order::print promises not to modify any data fields of Order. The programmer of the Order class uses const correctly and must rely on all other programmers to do the same.
Section 5.5: Default Constructors

- The purpose of a constructor is to initialize an object’s data fields. Some data fields that need initialization: data with `const`, references, numeric fields, etc.

- A default constructor has no parameters.

```cpp
class Product
{
public:
    Product();
...
};

Product::Product()
{
    price = 1;
    score = 0;
}
```

Here, `Product();` is a default constructor.
Section 5.6: Constructors with Parameters

class Employee
{
    public:
        Employee();
        Employee(string employee_name, double initial_salary);
        void set_salary(double new_salary);
        string get_name() const;
        double get_salary() const;
    private:
        string name;
        double salary;
};

Whenever two functions have the same name but are distinguished by their parameter types, the function name is overloaded. (See Advanced Topic 5.2 on page 248 for more information on overloading in C++.)
Section 5.7: Accessing Data Fields

- Private data fields can only be accessed by member functions of the same class.
- You should *not* automatically write accessor functions for *all* data fields. Sometimes it doesn’t make sense, and can cause user-created bugs, e.g. in the *Time* class.
Section 5.8: Comparing Member Functions with Nonmember Functions

Consider two functions that raise salary of the Employee class:

```cpp
void raise_salary(Employee& e, double percent)
{
    double new_salary = e.get_salary() * (1 + percent / 100);
    e.set_salary(new_salary);
}
...
raise_salary(harry, 7); // Raise Harrys salary by 7 percent
```

— and —

```cpp
class Employee
{
public:
    void raise_salary(double percent);
    ...;
};

void Employee::raise_salary(double percent)
{
    salary = salary * (1 + percent / 100);
}
...
...
raise_salary(harry, 7); // Raise Harrys salary by 7 percent
```

Which should you use? It depends on the ownership of the class. If you are designing class, then make useful operations into member functions. However, if you are using a class designed by someone else, then you should not add your own member functions. This is because the author of the class may make changes and improvements to the class.
Section 5.9: Separate Compilation

- The code of complex programs is distributed over multiple files.
- Header files contain:
  - Definitions of classes.
  - Declaration of constants.
  - Declaration of nonmember functions.
  - Declaration of global variables.
- Source files contain:
  - Definitions of member functions.
  - Definitions of nonmember functions.
  - Definitions of global variables.
Programming – Art or Science?

Both.
Disclaimer about slides

I claim no originality of the examples or instructional material above. Some I may have created, others I may have copied. I owe a tremendous thanks to those that have made their instructional materials available online.