PIC 10B Week 1 (Tues)

Alex Tong Lin

October 3, 2017

1. Loops
2. `const` keyword
3. Pass by reference v.s. pass by value
4. Constructor Initialization List
5. Functions: Member v.s. Non-member
6. Direct v.s. Indirect Initialization
7. Good coding practices
8. Classes

Loops

For Loop

```
for(i = 1; i < 10; i++) {
    cout << i << " ";
}
```

Output: 1 2 3 4 5 6 7 8 9

```
for(i = 1; i <= 10; i++) {
    cout << i << " ";
}
```

Output: 1 2 3 4 5 6 7 8 10

Loops

While Loop

```
int i;
i = 5;

while(i < 10) {
    cout << i << " ";
    i = i + 1;
}
```

Output: 5 6 7 8 9

Loops

Do...While Loop

If you want the loop to execute at least once.

```
int i;
i = 5;

do {
    cout << i << " ";
} while(i < 1);
```

Output: 5

Loops

Nested Loops

```
for(i = 1; i <= 5; i++) {

    for(j = 1; j <= i; j++) {
        cout << "[]";
    }

    cout << endl;
}
```

Output:

```
[]
[][]
[][][]
[][][][]
[][][][][]
```

const correctness

The const keyword declares that a variable is meant to be constant, i.e. not allowed to change. This is useful to the programmer to ensure your data is not destroyed.

```
int const pi = 3.1415926535;

pi = 3;
```

Output: Compiler error!

const correctness

```
void f(const int& y);

int main()
{
    int x = 1;
    f(x);
    cout << x << "\n";
}

void f(const int& y) {
    y = y + 1;
}
```

Output: Compiler error. “assignment of read-only reference ‘y’”

Passing by reference v.s. Passing by value

```
main() {
      int i = 10, j = 20;
      swapThemByVal(i, j);
      cout << i << "  " << j << endl;     // displays 10  20
      swapThemByRef(i, j);
      cout << i << "  " << j << endl;     // displays 20  10
   }

   void swapThemByVal(int num1, int num2) {
      int temp = num1;
      num1 = num2;
      num2 = temp;
   }

   void swapThemByRef(int& num1, int& num2) {
      int temp = num1;
      num1 = num2;
      num2 = temp;
   }
```

(Above example is thanks to Dr. Carol Zander, from University of Washington, Bothell)

Constructor Initialization List

```
class AA
{
    public:
    AA( int x )
    {
        cout << "AA's constructor " << "called with " << x << endl;
    }
};

class BB : public AA
{
        public:
        BB() : AA( 10 )  // construct the AA part of BB
        {
                cout << "BB's constructor" << endl;
        }
};

int main()
{
        BB thing;
}
```

Output: AA's constructor called with 10
BB's constructor

Constructor Initialization List

When do you want to use a constructor initialization list:

- When you have to pass a parameter to a parent constructor.
- When you have a field that is a `const`.
- When you have a field that is a reference. References are immutable, so they can only be initialized once.
- If one of the classes use does not have a default constructor.

Member functions v.s. Non-member functions

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

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

void set_values(Rectangle& rect) // non-member function
{
    rect.width = x;
    rect.height = y;
}

int main ()
{
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  set_values(5,6); // compiler error
  return 0;
}
```

Direct v.s. Indirect initialization

```
double a1(1.5); // direct initialization
double a2 = 1.5; // copy initialization
```

Direct v.s. Indirect initialization

Direct initialization is performed in the following situations:

1. initialization with a nonempty parenthesized list of expressions
2. during list-initialization sequence, if no initializer-list constructors are provided and a matching constructor is accessible, and all necessary implicit conversions are non-narrowing.
3. initialization of a prvalue temporary by functional cast or with a parenthesized expression list
4. initialization of a prvalue temporary by a static cast expression
5. initialization of an object with dynamic storage duration by a new-expression with a non-empty initializer
6. initialization of a base or a non-static member by constructor initializer list
7. initialization of closure object members from the variables caught by copy in a lambda-expression

Direct v.s. Indirect initialization

Copy initialization is performed in the following situations:

1. when a named variable (automatic, static, or thread-local) of a non-reference type T is declared with the initializer consisting of an equals sign followed by an expression.
2. (until C++11)when a named variable of a scalar type T is declared with the initializer consisting of an equals sign followed by a brace-enclosed expression (Note: as of C++11, this is classified as list initialization, and narrowing conversion is not allowed).
3. when passing an argument to a function by value
4. when returning from a function that returns by value
5. when throwing or catching an exception by value
6. as part of aggregate initialization, to initialize each element for which an initializer is provided

Good coding practices

(Will talk it out)

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

```
#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:

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

class Rectangle {
  private:
    int width, height;    // private members, only accessible within class
  public:
    void set_values(int,int);                       // 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

```
#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.

Section 5.2: Interface

Functions: Accessor v.s. Mutator, an example

```
#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);
}
```

Section 5.4: Member Functions

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();
...
};
...
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.

```
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:

```
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 —

```
class Employee
{
public:
    void raise_salary(double percent);
    ...
};
void Employee::raise_salary(double percent)
{
    salary = salary * (1 + percent / 100);
}
...
harry.raise_salary(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 have created, others I have copied. I owe a tremendous thanks to those that have made their instruction of C++ available online.