Programming Design

Classes (I)

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Object-oriented programming

- Until now, we have focused on **procedural programming**.
  - The keys are logical controls and subprocedures, i.e., `if`, `for`, and functions.
- We will begin to introduce a new programming philosophy: **object-oriented programming** (**OOP**).
  - It is based on procedural programming.
  - It is different in the perspective of thinking.
- In C, we use structures; in C++, we use **classes**.
- Like structures, we can use classes to define data types by ourselves.
  - When we create variables with classes, they are called **objects**.
- As we will see, classes are much more powerful than structures.

Outline

- **Motivations**
- Basic concepts
- Constructors and destructors
- Self-defined libraries

An example in `struct`

- Recall that we have the structure **`Point`** (which is a vector):

```
struct Point
{
  int x;
  int y;
  double distOri();
  void reflect()
};
```

```
double Point::distOri()
{
  return sqrt(pow(x, 2) + pow(y, 2));
}
void Point::reflect ()
{
  int temp = a.x;
  a.x = a.y;
  a.y = temp;
}
```

- May we generalize it into a **multi-dimensional** vector?

An example in `struct`

- Let's define a structure **`MyVector`**:

```
struct MyVector
{
  int n;
  int* m;
  void init(int dim);
};
void MyVector::init(int dim)
{
  n = dim;
  m = new int[n];
  for(int i = 0; i < n; i++)
    m[i] = 0;
}
```

```
int main()
{
  MyVector v;
  int dimension = 0;
  cin >> dimension;
  v.init(dimension);
  delete [] v.m;
  return 0;
}
```

An example in `struct`

- Let's add some member functions:

```
struct MyVector
{
  // old things
  void print();
};
void MyVector::print()
{
  cout << "(";
  for(int i = 0; i < n - 1; i++)
    cout << m[i] << ", ";
  cout << m[n-1] << ")\n";
}
```

```
int main()
{
  MyVector v;
  v.init(5);
  v.m[0] = 10;
  v.print();
  delete [] v.m;
  return 0;
}
```

Drawbacks for using a structure

- Several drawbacks:
  - We may forget to initialize the vector.
  - Another programmer may print a vector in a bad way.
  - **n** and the length of the dynamic array **m** may be inconsistent.
  - We may forget to release the spaces allocated dynamically.

```
MyVector v;
v.print();
delete [] v.m;
```

```
MyVector v;
v.init(5);
v.m[0] = 10;
cout << "(";
for(int i = 0; i < n - 1; i++)
  cout << m[i] << ", ";
cout << m[n-1];
delete [] v.m;
```

```
MyVector v;
int dim = 0;
cin >> dim;
v.init(dim);
cin >> v.n;
delete [] v.m;
```

```
MyVector a;
int dim = 0;
cin >> dim;
a.init(dim);
```

Drawbacks for using a structure

- Our hopes:
  - The initializer can be called automatically.
  - The vector can be printed only in allowed ways.
  - `n` and the length of the dynamic array `m` cannot be modified separately.
  - Spaces allocated dynamically will be released automatically.
- These issues may are not apparent when the program is of a small scale.
  - They emerge when **multiple programmers** collaborate in one project.
  - They emerge when you revise a program that you wrote **three months ago**.

Drawbacks for using a structure

- So we use classes in C++!
- Recall our hopes:
  - The initializer can be called automatically.
  - The vector can be printed only in allowed ways.
  - `n` and the length of the dynamic array `m` cannot be modified separately.
  - Spaces allocated dynamically will be released automatically.
- In C++, a class can:
  - Define member functions that will **be called automatically** when and only when an object is created/destroyed.
  - **Hide some members** and open only allowed members to the public.
  - And many more.

Instance vs. static variables/functions

- In a class, we can define variables and functions, just like in a structure.
  - They are call **member variables** and **member functions**.
- However, now there can be four types of class members:
  - **Instance variables** (default).
  - **Static variables**.
  - **Instance functions** (default).
  - **Static functions**.
- Starting from now, when we say member variables (fields) and member functions, we are talking about instance ones.

Outline

- Motivations
- **Basic concepts**
- Constructors and destructors
- Self-defined libraries

Class definition

- To define a class:
  - Simply change **`struct`** to **`class`**.
  - We may also define the function inside the class definition block.
- Compilation error! Why?

```
int main()
{
  MyVector v;
  v.init(5);
  delete [] v.m;
  return 0;
}
```

```
void MyVector::print()
{
  cout << "(";
  for(int i = 0; i < n - 1; i++)
    cout << m[i] << ", ";
  cout << m[n-1] << ")\n";
}
```

```
class MyVector
{
  int n;
  int* m;
  void init(int dim);
  void print();
};
void MyVector::init(int dim)
{
  n = dim;
  m = new int[n];
  for(int i = 0; i < n; i++)
    m[i] = 0;
}
```

Visibility

- We can/must set visibility of members in a class:
  - **Public** members can be accessed **anywhere**.
  - **Private** members can be accessed only **in the class**.
  - **Protected** members will be discussed later in this semester.
- These three keywords are the **visibility modifiers**.
- By **default**, all members' visibility level is **private**.
  - That is why `v.init(5)` generates a compilation error; `init()` is private and cannot be invoked outside the class (e.g., in the main function).
- By setting visibility, we can **hide**/**open** our instance members.
  - Usually all instance variables are private.
  - Let's see how to do this.

Visibility

- A class with different visibility levels:
- Private instance members can only be accessed **inside** the **definition** of **instance functions**.
  - E.g., **`init()`** and **`print()`**.
- Public instance members can be accessed everywhere.

```
class MyVector
{
private:
  int n;
  int* m;
public:
  void init(int dim);
  void print();
};
```

```
int main()
{
  MyVector v;
  v.init(5); // OK!
  delete [] v.m;
  return 0;
}
```

Why data hiding?

- Setting members to private is to do **data hiding**.
- Why bother?
- By setting members to private, we **control** the way that they are accessed.
  - We can better predict how others may use our class.
- As an example, now we can prevent inconsistency between **n** and the length of **m**!

```
int main()
{
  MyVector v;
  v.init(5); // fine
  v.n = 3; // compilation error!
  delete [] v.m;
  return 0;
}
```

Why data hiding?

- As another example, we do not want a vector to be printed out in strange formats, such as {0, 10, 20}, [0, 10, 20), (0-10-20), etc.
  - We want they all look the same, like (5, 6, 7).
  - If we allow other programmers to access **n** and **m**, they can print out a vector in any way they like!
  - So we privatize instance variables and **provide** a public member function **print()** to control (restrict) the way of printing a vector.
- These public member functions are often called **interfaces**. All others should communicate with the class through interfaces.

```
class MyVector
{
private:
   int n;
   int* m;
public:
   void init(int dim);
   void print();
};
```

Visibility

- In general, some instance variables/functions should not be accessed directly (or even known) by other ones.
  - They should be used only in the class.
  - In this case, set them private.
- You may see many classes with all instance variables private and all instance functions public.
  - If you do not know what to do, do this.
  - However, any instance function that **should not be invoked by others** should also be private.

Private instance functions

- In an instance function, we can invoke an instance function.
- Set an instance function private if it should be not accessed by others.

```
class MyVector {
private:
  int n;
  int* m;
  int max();
public:
  void init(int dim);
  void print();
  void printMax();
};
```

```
int MyVector::max() {
  int max = m[0];
  for(int i = 1; i < n; i++) {
    if(m[i] > max)
      max = m[i];
  }
  return max;
}
void MyVector::printMax() {
  cout << "Max: " << max() << "\n";
}
```

Encapsulation

- The concepts of **packaging** (grouping member variables and member functions) and **data hiding** together form the concept of "**encapsulation**".
  - Roughly speaking, we pack data (member variables) into a **black box** and provide only **controlled interfaces** (member functions) for others to access these data.
  - Others should not even know how those interfaces are implemented.
- For OOP, there are three main characteristics/functionalities:
  - **Encapsulation**.
  - **Inheritance**.
  - **Polymorphism**.
- The last two will be discussed later in this semester.

Instance function overloading

- We can **overload** an instance function with different parameters.

```
class MyVector
{
private:
  int n;
  int* m;
public:
  void init();
  void init(int dim);
  void init(int dim, int value);
  void print();
};
```

```
void MyVector::init()
{
  n = 0;
  m = NULL;
}
void MyVector::init(int dim)
{
  init(dim, 0);
}
void MyVector::init(int dim, int value)
{
  n = dim;
  m = new int[n];
  for(int i = 0; i < n; i++)
    m[i] = value;
}
```

Objects as arguments or return values

- We can pass an object into any function.
- A function can return an object.
- `MyVector add(MyVector v1, MyVector v2);`
  - Returns the sum of the two input vectors.
  - This should be a global function rather than an instance function. Why?

Objects as instance variables

- An instance variable's type can be a class.
- In other words, an object can **have other objects as members**.
  - This can also happen for structures.
- For example:

```
class MyTriangle
{
private:
  MyVector vertex1;
  MyVector vertex2;
  MyVector vertex3;
  // ...
};
```

```
class MyPolytope
{
private:
  int n; // number of vertices
  MyVector* vertex;
  // ...
};
```

Outline

- Motivations
- Basic concepts
- **Constructors and destructors**
- Self-defined libraries

Our hopes

- Recall our hopes:
  - The initializer can be called automatically.
  - The vector can be printed only in allowed ways.
  - **n** and the length of the dynamic array **m** cannot be modified separately.
  - Spaces allocated dynamically will be released automatically.
- The second and third have been done.
- The first and the last require **constructors** and **destructors**.

```
class MyVector
{
private:
  int n;
  int* m;
public:
  void init();
  void init(int dim);
  void init(int dim, int value);
  void print();
};
```

Constructors

- A constructor is an **instance function** of a class.
  - However, it is very special.
- A constructor will be invoked **automatically** when the object is **created**.
  - It must be invoked.
  - It cannot be invoked twice.
  - It cannot be invoked by the programmer manually.
- Usually it is used to initialize the object.

Constructors

- A constructor's name is **the same as** the class.
- It does not return anything, not even **void**.
- You can (and usually will) overload them.
- The constructor with **no parameter** is the **default constructor**.
- If, and only if, a programmer does not define any constructor, the **compiler** makes a default one which **does nothing**.
- A constructor may be private.
  - Be invoked only by other constructors.

```
class MyVector
{
private:
  int n;
  int* m;
public:
  MyVector();
  MyVector(int dim);
  MyVector(int dim, int value);
  void print();
};
```

Constructors for MyVector

- Let's define our class **MyVector** with constructors:

```
class MyVector
{
private:
  int n;
  int* m;
public:
  MyVector();
  MyVector(int dim, int value = 0);
  void print();
};
```

```
MyVector::MyVector()
{
  n = 0;
  m = NULL;
}
MyVector::MyVector(int dim, int value)
{
  n = dim;
  m = new int[n];
  for(int i = 0; i < n; i++)
    m[i] = value;
}
```

- Just like usual functions, a constructor may have a default argument.

Constructors for `MyVector`

- Now, in the main function, we assign initial values when we declare objects:

```
int main()
{
  MyVector v1(1);
  MyVector v2(3, 8);
  v1.print(); // (0)
  v2.print(); // (8, 8, 8)
  return 0;
}
```

- If any member variable needs an initial value when an object is created, you should write a constructor to initialize it.
- Use constructor overloading to provide flexibility.

Destructors

- A destructor is invoked right before an object is **destroyed**.
  - It must be public and have no parameter.
- The compiler provides a default destructor that does nothing.
- To define your own destructor, use **~**:

```
class MyVector
{
  // ...
public:
  // ...
  ~MyVector() { cout << "Bye~\n"; }
};
```

Why destructors?

- Suppose we do not define our own destructor.
- Then there may be memory leak when an object is destroyed.
  - When there is **dynamic memory allocation**.
  - Typically when there is a pointer member.

```
int main()
{
  for(int i = 0; i < 10; i++) {
    MyVector v1(1);
    // memory leak
  }
  return 0;
}
```

```
MyVector::MyVector
   (int dim, int value)
{
  n = dim;
  m = new int[n];
  for(int i = 0; i < n; i++)
    m[i] = value;
}
```

Why destructors?

- One typical mission for a destructor is to release those **dynamically allocated memory spaces** pointed by member variables.
  - The default destructor does not do this. We must do this by ourselves.

```
int main()
{
  for(int i = 0; i < 10; i++) {
    MyVector v1(1);
    // no memory leak!
  }
  return 0;
}
```

```
class MyVector
{
private:
  int n;
  int* m;
public:
  // ...
  ~MyVector()
  {
    delete [] m;
  }
};
```

Timing for constructors/destructors

- When a class has other classes as types of instance variables, when are all the constructors/destructors invoked?

```
int main()
{
  B b;
  return 0;
}
```

```
class A
{
public:
  A() { cout << "A\n"; }
  ~A() { cout << "a\n"; }
};
class B
{
private:
  A a;
public:
  B() { cout << "B\n"; }
  ~B() { cout << "b\n"; }
};
```

Outline

- Motivations
- Basic concepts
- Constructors and destructors
- **Self-defined libraries**

Libraries

- There are many C++ standard **libraries**.
  - **<iostream>**, **<climits>**, **<cmath>**, **<cctype>**, **<cstring>**, etc.
  - Many (constant) variables and functions are defined there.
  - Many more.
- We may also want to define **our own libraries**.
  - Especially when we collaborate with others.
  - Typically, one implements classes or global functions for the others to use.
  - That function can be defined in a self-defined library.
- A library includes a **header file** (.h) and a **source file** (.cpp).
  - The header file contains declarations; the source file contains definitions.

Example

- Consider the following program with a single function **myMax()**:

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

int myMax (int [], int);
int main ()
{
  int a[5] = {7, 2, 5, 8, 9};
  cout << myMax (a, 5);
  return 0;
}
```

```
int myMax (int a[], int len)
{
  int max = a[0];
  for (int i = 1; i < len; i++)
  {
    if (a[i] > max)
      max = a[i];
  }
  return max;
}
```

- Let's define a constant **variable** for the array length in **a header file**.

Defining variables in a library

myMax.h

```
const int LEN = 5;
```

main.cpp

```
#include <iostream>
#include "myMax.h"
using namespace std;

int myMax (int [], int);
int main ()
{
  int a[LEN] = {7, 2, 5, 8, 9};
  cout << myMax (a, LEN);
  return 0;
}
```

```
int myMax (int a[], int len)
{
  int max = a[0];
  for (int i = 1; i < len; i++)
  {
    if (a[i] > max)
      max = a[i];
  }
  return max;
}
```

Including a header file

- When your main program wants to include a self-defined header file, simply indicate its path and file name.
  - `#include "myMax.h"`
  - `#include "D:/test/myMax.h"`
  - `#include "lib/myMax.h"`
  - Using `\` or `/` does not matter (on Windows).
- We still compile the main program as usual.
- Let's also define **functions** in our library!
  - Now we need a source file.

Defining functions in a library

myMax.h

```
const int LEN = 5;
int myMax(int [], int);
```

main.cpp

```
#include <iostream>
#include "myMax.h"
using namespace std;

int main ()
{
  int a[LEN] = {7, 2, 5, 8, 9};
  cout << myMax(a, LEN);
  return 0;
}
```

myMax.cpp

```
int myMax(int a[], int len)
{
  int max = a[0];
  for (int i = 1; i < len; i++)
  {
    if (a[i] > max)
      max = a[i];
  }
  return max;
}
```

Including a header and a source file

- When your main program also wants to include a self-defined source file, the include statement needs not be changed.
  - **`#include "myMax.h"`**
- We add a source file myMax.cpp.
  - In the source file, we **implement** those functions declared in the header file.
  - The main file names of the header and source files can be different.
- The two source files (main.cpp and myMax.cpp) must be **compiled together**.
  - Each environment has its own way.
  - In Dev-C++, we simply create a “console project”.

Defining one more function

myMax.h

```
const int LEN = 5;
int myMax (int [], int);
void print(int);
```

main.cpp

```
#include <iostream>
#include "myMax.h"
using namespace std;

int main ()
{
  int a[LEN] = {7, 2, 5, 8, 9};
  print(myMax(a, LEN));
  return 0;
}
```

myMax.cpp

```
int myMax(int a[], int len)
{
  int max = a[0];
  for (int i = 1; i < len; i++)
  {
    if (a[i] > max)
      max = a[i];
  }
  return max;
}
void print(int i)
{
  cout << i; // cout undefined!
}
```

Defining one more function

- Each source file contains statements to run.
- Each source file must include the libraries it needs for its statements.

```
#include <iostream>
using namespace std;
int myMax (int a[], int len)
{
  int max = a[0];
  for (int i = 1; i < len; i++)
  {
    if (a[i] > max)
      max = a[i];
  }
  return max;
}
void print (int i)
{
  cout << i; // good!
}
```

The complete set of files

myMax.h

```
const int LEN = 5;
int myMax (int [], int);
void print (int);
```

main.cpp

```
#include <iostream>
#include "myMax.h"
using namespace std;

int main ()
{
  int a[LEN] = {7, 2, 5, 8, 9};
  print (myMax (a, LEN));
  return 0;
}
```

myMax.cpp

```
#include <iostream>
using namespace std;
int myMax (int a[], int len)
{
  int max = a[0];
  for (int i = 1; i < len; i++)
  {
    if (a[i] > max)
      max = a[i];
  }
  return max;
}
void print (int i)
{
  cout << i;
}
```

Remarks

- In many cases, myMax.cpp also include myMax.h.
  - E.g., if `LEN` is accessed in myMax.cpp.
- More will be discussed in further courses (e.g., Data Structures).
  - More than two source files.
  - A header file including another header file.