IM 1003: Programming Design

Classes (I)

Ling-Chieh Kung

Department of Information Management
National Taiwan University

April 7, 2014

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 methodology: **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

An example in `struct`

- Recall that we have the 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
{
  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;
}
```

```
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.
  - The vector may be printed 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.

```
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

- 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 its members and open only allowed members to the public.

Instance vs. static variables/functions

- In a class, we can define variables and functions, just as we did in a structure.
  - They are call **member variables** and **member functions**.
- However, now there are 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

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`**: it can be accessed **anywhere**.
  - **`private`**: it can be accessed only **in the class**.
  - **`protected`**: discussed later in this semester.
- These three keywords are the **visibility modifiers**.
- The default visibility level is **`private`**.
  - That is why **`v.init(5)`** in the main function generates a compilation error.
- By setting visibility, we can **hide** 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()`**.
- Once we set **`n`** and **`m`** as private:
  - It is impossible for the vector to be printed in a bad way.
  - It is impossible for **`n`** and the size of **`m`** to be inconsistent!

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

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

Invoking instance functions in classes

- In an instance function, we can invoke an instance function.

```
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";
}
```

Why data hiding?

- In general, when we write a class, we want it to work **as we expect**.
  - That is, "**under control**".
- For 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.
  - If we allow another programmer to access **`n`** and **`m`** in their programs, he can print out a vector in any way he likes!
  - So we set instance variables to be private and make **`print()`** public.
- Similarly, setting **`n`** and **`m`** private and leaving **`print()`** public present inconsistency between **`n`** and the size of **`m`**.

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.

Encapsulation

- The concept of **packaging** (member variables and member functions) and **data hiding** is together called "**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.
- `Vector 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**

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 the 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, 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.

```
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;
}
```

Constructors for MyVector

- Now, in the main function:

```
int main()
{
  MyVector v1(1);
  MyVector v2(3, 8);
  v1.print(); // (0)
  v2.print(); // (8, 8, 8)
  // memory leak
  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.
- To replace the default destructor by a self-defined one, use **~**:

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

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. One must do this by herself/himself.
  - If this is not done, there will be memory leaks.

```
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"; }
};
```