Programming Design

Classes

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**.
- Using classes properly enhances modularity and makes large-scale system design and development easier.

Outline

- **Basic concepts**
- Constructors and the destructor
- Friends and static members
- Objects pointers and the copy constructor

An example

- Recall that we have the structure **`Point`** (which is a two-dimensional vector).
- Let's implement a **multi-dimensional** vector:

```
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(3);
  v.m[0] = 3;
  v.print(); // (3, 0, 0)
  delete [] v.m;
  return 0;
}
```

Some drawbacks

- We may forget to initialize the vector.
- Another programmer may print out 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(3);
cout << "<";
for(int i = 0; i < 3; i++)
  cout << m[i] << "-";
cout << m[n-1] << "]";
// <3-0-0]
delete [] v.m;
```

```
MyVector v;
int dim = 3;
v.init(dim);
v.n = 6;
delete [] v.m;
```

```
MyVector a;
int dim = 0;
cin >> dim;
a.init(dim);
// no delete
```

Some drawbacks

- 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 emerge when **multiple programmers** collaborate in one project.
- 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 **member variables** and **member functions**:
  - **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.

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;
}
```

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

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**.
- 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;
}
```

```
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";
}
```

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.
- Therefore,
  - Now we can prevent inconsistency between **n** and the length of **m**.
  - We can prevent a vector from being printed out in strange formats, such as {0, 10, 20}, [0, 10, 20), (0-10-20), etc.

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

- Public member functions are often called **interfaces**.
  - All others should communicate with the class through interfaces.

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 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 = nullptr;
}
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 for functions and class members

- We can pass an object into any function and/or return an object.

```
MyVector add(MyVector v1, MyVector v2);
```

- An instance variable's type can be a class.

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

```
class MyPolytope
{
private:
  int vertexCount;
  MyVector* vertex;
  // ...
};
```

Outline

- Basic concepts
- **Constructors and the destructor**
- Friends and static members
- Objects pointers and the copy constructor

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(); // constructors
  MyVector(int dim);
  MyVector(int dim, int value);
  void print();
};
```

Constructors for MyVector

- Let's define our class **MyVector** with constructors:
  - Just like usual functions, a constructor may have a default argument.

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

```
MyVector::MyVector()
{
  n = 0;
  m = nullptr;
}
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, 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 **~**.
  - Typically we release dynamically allocated space in a destructor.

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

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

int main()
{
  if (true)
    MyVector v1(1);
    // no memory leak
  return 0;
}
```

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

- Basic concepts
- Constructors and the destructor
- **Friends and static members**
- Objects pointers and the copy constructor

Getters and setters

- In most cases, instance variables are private.
- For them to be accessed, sometimes people implement **getters** and **setters** for them.
  - A getter simply returns the value of a private instance variable.
  - A setter simply modifies a private instance variables to a given value.
- What are the benefits and **costs** for having getters and setters?

```
class MyVector
{
private:
  int n;
  int* m;
public:
  // ...
  int getN() { return n; }
  void setN(int v) { n = v; }
};
```

`friend` for functions and classes

- To "open" private members, another way is to declare "**friends**."
- One class can allow its friends to access its private members.
- Its friends can be **global functions** or other **classes**.
  - Then inside **`test()`** and member functions of **`Test`**, those private members of **`MyVector`** can be accessed.
  - **`MyVector`** cannot access **`Test`**'s members.
- A friend can be declared in either the public or private section. It does not matter.
- A class must declare its friends **by itself**.
  - One cannot declare itself as another one's friend!

```
class MyVector
{
  // ...
friend void test();
friend class Test;
};
```

friend: an example

```
void test() {
  MyVector v;
  v.n = 100; // syntax error if not a friend
  cout << v.n; // syntax error if not a friend
}
```

```
class Test {
public:
  void test(MyVector v) {
    v.n = 200; // syntax error if not a friend
    cout << v.n; // syntax error if not a friend
  }
};
```

`friend` for functions and classes

- Declare friends only if data hiding is preserved.
  - Do not set everything public!
  - Use structures rather than classes when nothing should be private (this is recommended but not required).
  - Be careful in offering public member functions (e.g., getters and setters).
- **`friend`** in fact **help you hide data**.
  - If a private member should be accessed only by another class/function, we should declare a friend instead of writing a getter/setter.

Static members

- A class contains some instance variables and functions.
  - Each object has its own copy of instance variables and functions.
- A member variable/function may be an attribute/operation **of a class**.
  - When the attribute/operation is **class-specific** rather than object-specific.
  - A class-specific attribute/operation should be identical for all objects.
- These variables/functions are called **static members**.

Static members: an example

- In MS Windows, each window is an object.
  - Windows is written in C++.
  - Mac OS is written in Objective-C.
- Each window has some object-specific attributes.
- They also share one class-specific attribute: the color of their title bars.

```
class Window
{
private:
  int width;
  int height;
  int locationX;
  int locationY;
  int status; // 0: min, 1: usual, 2: max
  static int barColor; // 0: gray, ...
  // ...
public:
  static int getBarColor();
  static void setBarColor(int color);
  // ...
};
```

Static members: an example

- We have to initialize a static variable **globally**.

```
int Window::barColor = 0; // default

int Window::getBarColor()
{
  return barColor;
}

void Window::setBarColor(int color)
{
  barColor = color;
}
```

- To access static members, use ***class name*::*member name***.

```
int main()
{
  Window w; // not used
  cout << Window::getBarColor();
  cout << "\n";
  Window::setBarColor(1);
  return 0;
}
```

Static members

- Recall that we have four types of members:
  - Instance variables and instance functions.
  - Static variables and static functions.
- Some rules regarding static members:
  - We **may** access a static member inside an instance function.
  - We **cannot** access an instance member inside a static function.
  - Though **not suggested**, we **may** access a static member through an object.

```
Window w;
cout << w.getBarColor() << "\n";
```

Good programming style

- If one attribute should be identical for all objects, it should be declared as a static variable.
  - Do not make it an instance variable and try to maintain consistency.
- Do not use an object to invoke a static member.
  - This will confuse the reader.
- Use ***class name*****::*****member name*** even inside member function definition to show that it is a static member.

```
int Window::getBarColor()
{
  return Window::barColor;
}
```

Another way of using static members

- One may use a static global variable to count the number of times a global function is invoked.
- One may use a **static member variable** to count for how many times **an object is created**.

```
class A
{
private:
  static int count;
public:
  A() { A::count++; }
  static int getCount()
  { return A::count; }
};
```

```
int A::count = 0;

int main()
{
  A a1, a2, a3;
  cout << A::getCount() << "\n"; // 3
  return 0;
}
```

Another way of using static members

- With the help of the destructor, we may keep a record on the number of **active** (**alive**) objects.

```
class A
{
private:
  static int count;
public:
  A() { A::count++; }
  ~A() { A::count--; }
  static int getCount()
  { return A::count; }
};
```

```
int A::count = 0;

int main()
{
  if(true)
    A a1, a2, a3;
  cout << A::getCount() << "\n"; // 0
  return 0;
}
```

Outline

- Basic concepts
- Constructors and the destructor
- Friends and static members
- **Object pointers and the copy constructor**

Object pointers

- A class is a (self-defined) data type.
- A pointer may point to any data type.
  - A pointer may point to an **object**, i.e., store the address of an object.
- For example:

```
int main()
{
  MyVector v(5);
  MyVector* ptrV = &v; // object pointer
  return 0;
}
```

Object pointers

- What we have done is to use an object to invoke instance functions.
  - E.g., **`a.print()`** where **`a`** is an object and **`print()`** is an instance function.
- If we have a pointer **`ptrA`** pointing to the object **`a`**, we may write **`(*ptrA).print()`** to invoke the instance function **`print()`**.
  - **`*ptrA`** returns the object **`a`**.
- To simplify this, C++ offers the member access operator **`->`**.
  - This is specifically for an object pointer to access its members.
  - **`(*ptrA).print()`** is **equivalent** to **`ptrA->print()`**.

```
int main()
{
  MyVector v(5);
  MyVector* ptrV = &v;
  v.print();
  ptrV->print();
  return 0;
}
```

Why object pointers?

- Object pointers can be more useful than pointers for basic data types. Why?
- When one creates an array of objects, only the **default constructor** may be invoked.
  - Creating an array of object pointers delays the invocation of constructors.
  - These pointers than point to **dynamically** allocated objects.
- Passing a pointer into a function can be **more efficient** than passing the object.
  - A pointer can be much **smaller** than an object.
  - Copying a pointer is easier than **copying an object**.
- Other reasons will be discussed in other lectures.

Static object arrays

- We may also create object arrays.
  - The **default constructor** will be invoked.
  - There is no way to invoke other constructors.
  - We must implement other functions to assign proper values to instance variables.

```
int main()
{
  MyVector v[3]; // an object array
  v[0].print(); // run-time error!
  return 0;
}
```

Dynamic object arrays

- Object pointers allow us to do dynamic memory allocation.

```
int main()
{
  MyVector* ptrV = new MyVector(5);
  ptrV->print();
  delete ptrV;
  return 0;
}
```

- Object pointers allow us to create dynamic arrays.

```
int main()
{
  MyVector* ptrV = new MyVector[5];
  ptrV[0].print(); // run-time error
  delete [] ptrV;
  return 0;
}
```

Object pointer arrays

- To **delay** the invocation of constructors, we create an **object pointer array**.
  - Each pointer then points to a **dynamic object**.

```
int main()
{
  MyVector* ptrArray[5]; // no constructor invocation
  for(int i = 0; i < 5; i++)
    ptrArray[i] = new MyVector(i + 1); // constructor
  ptrArray[0]->print(); // (0)
  // some delete statements
  return 0;
}
```

Passing objects into a function

- Consider a function that takes three vectors and returns their sum.

```
MyVector sum
   (MyVector v1, MyVector v2, MyVector v3)
{
  // assume that their dimensions are identical
  int n = v1.getN();
  int* sov = new int[n];
  for(int i = 0; i < n; i++)
    sov[i] = v1.getM(i) + v2.getM(i) + v3.getM(i);
  MyVector sumOfVec(n, sov);
  return sumOfVec;
}
```

```
int MyVector::getN()
{ return n; }
int MyVector::getM(int i)
{ return m[i]; }
MyVector::MyVector
   (int d, int v[])
{
  n = d;
  for(int i = 0; i < n; i++)
    m[i] = v[i];
}
```

  - We need to create **four** **`MyVector`** objects in this function.

Passing object pointers into a function

- We may **pass pointers** rather than objects into this function:

```
MyVector sum(MyVector* v1, MyVector* v2, MyVector* v3)
{
  // assume that their dimensions are identical
  int n = v1->getN();
  int* sov = new int[n];
  for(int i = 0; i < n; i++)
    sov[i] = v1->getM(i) + v2->getM(i) + v3->getM(i);
  MyVector sumOfVec(n, sov);
  return sumOfVec;
}
```

  - We need to create **only one** **`MyVector`** object in this function.
  - Nevertheless, using pointers to access members requires more time.

Passing object references

- We may also **pass references**:

```
MyVector sum(MyVector& v1, MyVector& v2, MyVector& v3)
{
  // assume that their dimensions are identical
  int n = v1.getN();
  int* sov = new int[n];
  for(int i = 0; i < n; i++)
    sov[i] = v1.getM(i) + v2.getM(i) + v3.getM(i);
  MyVector sumOfVec(n, sov);
  return sumOfVec;
}
```

  - We create **only one** **`MyVector`** object in this function.

Constant references

- While we may want to pass references to save time, we need to protect our arguments from being modified.

```
MyVector sum
  (const MyVector& v1, const MyVector& v2, const MyVector& v3)
{
  // ...
}
```

  - Save time while **being safe**!
- Should we do the same thing when passing object pointers?

Copying an object

- Consider the following program:

```
class A
{
private:
  int i;
public:
  A() { cout << "A"; }
};
void f(A a1, A a2, A a3)
{
  A a4;
}
```

```
int main()
{
  A a1, a2, a3; // AAA
  cout << "\n===\n";
  f(a1, a2, a3); // A
  return 0;
}
```

- Why just one "**A**" when invoking **f()**?

Copying an object

- In general, when we pass by value, a local variable will be created.
  - When we pass by value for an object, a local object is created.
  - The constructor should be invoked.
  - So why just one "**A**" when invoking **f()**?
- How about this?
  - No constructor is invoked when **a4** is created?

```
int main()
{
  A a1, a2, a3; // AAA
  cout << "\n===\n";
  A a4 = a1; // nothing!
  return 0;
}
```

Copying an object

- Creating an object by "copying" an object is a special operation.
  - When we pass an object into a function using the call-by-value mechanism.
  - When we assign an object to another object.
  - When we create an object with another object as the argument of the constructor.

```
f(a1, a2, a3);
```

```
A a4 = a1;
```

```
A a5(a1);
```

- When this happens, the **copy constructor** will be invoked.
  - If the programmer does not define one, the compiler adds a **default copy constructor** (which of course does not print out anything) into the class.
  - The default copy constructor simply copies all member variables one by one, regardless of the variable types.

Copy constructors

- We may implement our own copy constructor.
  - In the C++ standard, the parameter must be a **constant reference**.
  - If calling by value, it will invoke itself infinitely many times.

```
class A
{
private:
  int i;
public:
  A() { cout << "A"; }
  A(const A& a) { cout << "a"; }
};
```

```
void f(A a1, A a2, A a3)
{
  A a4;
}
int main()
{
  A a1, a2, a3; // AAA
  cout << "\n===\n";
  f(a1, a2, a3); // aaaA
  return 0;
}
```

Copy constructors for `MyVector`

- For **`MyVector`**, one way to implement a copy constructor is
  - This has nothing different from the default copy constructor.
  - If no member is an array/pointer, the default copy constructor is fine.
- If there is any array or pointer member variable, the default copy constructor does "**shallow copy**".
  - And two different vectors may share the same space for values.
  - Modifying one vector affects the other!

```
MyVector::MyVector
   (const MyVector& v)
{
  n = v.n;
  m = v.m;
}
```

```
int main()
{
  MyVector v1(5, 1);
  MyVector v2(v1); // what is bad?
}
```

Deep copy

- To correctly copy a vector (by creating new values), we need to write our own copy constructor.
- We say that we implement "**deep copy**" by ourselves.
  - In the self-defined copy constructor, we **manually create another dynamic array**, set its elements' values according to the original array, and use **m** to record its address.

```
MyVector::MyVector(const MyVector& v)
{
  n = v.n;
  m = new int[n]; // deep copy
  for(int i = 0; i < n; i++)
    m[i] = v.m[i];
}
```