Programming Design

Classes (II)

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

- **Static members**
- Objects and pointers
- **`friend`**, **`this`**, and **`const`**

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

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

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

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

```
int main()
{
  Window w; // not used
  cout << Window::getBarColor();
  cout << endl;
  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() << endl;
```

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() << endl; // 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() << endl; // 0
  return 0;
}
```

Outline

- Static members
- **Objects and pointers**
- `friend`, `this`, and `const`

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.
- Recall the class **MyVector**:

```
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()`**.
  - **`(*ptrA).x`** is equivalent to **`ptrA->x`**.

Object pointers

- An example of using an object pointer:
  - `new MyVector(5)` dynamically allocates a memory space.

```
int main()
{
  // an object pointer
  MyVector* ptrV = new MyVector(5);
  // instance function invocation
  ptrV->print();
  delete ptrV;
  return 0;
}
```

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

Why object pointers?

- Object pointers are more useful than pointers for basic data types. Why?
- Passing a pointer into a function is **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.

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 cenGrav(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 cenGrav
  (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**, we may implement a copy constructor as:

```
MyVector::MyVector(const MyVector& v)
{
  n = v.n;
  m = v.m; // copying the address in v.m to m
}
```

  - This has nothing different from the default copy constructor.

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

Shallow copy

- If no member variable 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; // shallow copy
}
```

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];
}
```

Outline

- Static members
- Objects and pointers
- **friend, this, and const**

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.
  - Write appropriate public member functions (e.g., getters and setters).
- **friend** may also 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.

this

- When you create an object, it occupies a memory space.
- Inside an instance function, **this** is a **pointer** storing the address of that object.
  - **this** is a C++ keyword.
- When the compiler reads **this**, it looks at the memory space to find the object.
- The two implementations are identical:

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

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

this

- Suppose that **x** is an instance variable.
  – Usually you can use **x** directly instead of **this->x**.
  – However, if you want to have a **local variable** or **function parameter** having the same name as an instance variable, you need **this->**.

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

```
MyVector::MyVector(int n, int m[])
{
  this->n = n;
  for(int i = 0; i < n; i++)
    this->m[i] = m[i];
}
```

- A local variable hides the instance variable with the same name.
  – **this->x** is the instance variable and **x** is the local variable.

Good programming style

- You may choose to always use **`this->`** when accessing instance variables and functions.
- This will allow other programmers (or yourself in the future) to know they are members without looking at the class definition.

Constant objects

- Some variables are by nature **constants**.

```
const double PI = 3.1416;
```

- We may also have **constant objects**.

```
const MyVector ORIGIN_3D(3, 0);
```

  – This is the origin in $\mathbf{R}^3$. It should not be modified.
- Should there be any restriction on **instance function invocation**?

Constant objects

- A constant object cannot invoke a function that modifies its instance variables.
  - In C++, functions that may be invoked by a constant object must be declared as a **constant instance function**.
- For a constant instance function:
  - It can be called by non-constant objects.
  - It cannot modify any instance variable.
- For a non-constant instance function:
  - It cannot be called by constant objects even if no instance variable is modified.

```
class MyVector
{
private:
  int n;
  int* m;
public:
  MyVector();
  MyVector(int dim, int v[]);
  ~MyVector();
  int getN() const;
  int getM() const;
  void print();
};
```

Constant instance variables

- We may have **constant instance variables**.
  - E.g., for a vector, its dimension should be fixed once it is determined.
- Obviously, a constant instance variable should be initialized in the constructor(s).
  - However:

```
MyVector::MyVector()
{
  n = 0; // error!
  m = NULL;
}
```

```
class MyVector
{
private:
  const int n;
  int* m;
public:
  MyVector();
  MyVector(int dim, int v[]);
  ~MyVector();
  int getN() const;
  int getM() const;
  void print();
};
```

Member initializers

- For a constant instance variable:
  - It cannot be assigned a value.
  - It cannot be initialized globally.
- We need a **member initializer**.
  - A specific operation for initializing an instance variable.
  - Can also be used for initializing non-constant instance variables.

```
class MyVector
{
private:
  const int n;
  int* m;
public:
  MyVector() : n(0) { m = NULL; }
  MyVector(int dim, int v[]) : n(dim)
  {
    for(int i = 0; i < n; i++)
      m[i] = v[i];
  }
  // ...
};
```

Initializing constant instance variables

- Member initializers can also be used when constructors are implemented outside the class definition block.

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

```
class MyVector
{
private:
  const int n;
  int* m;
public:
  MyVector();
  MyVector(int dim, int v[]);
  // ...
};
```

- Member initializers are used a lot in general.