

IM 1003: Programming Design

Inheritance

Ling-Chieh Kung

Department of Information Management
National Taiwan University

May 12, 2014

Outline

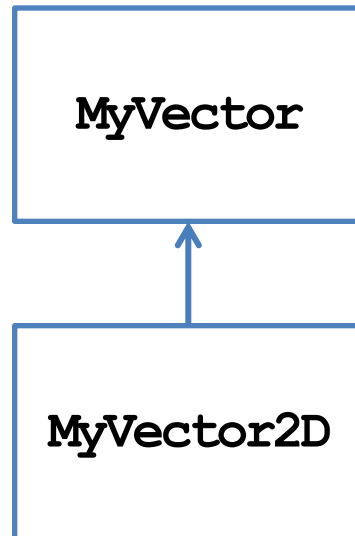
- **Basic ideas and the first example**
- Defining new members and overriding old members
- Cascade inheritance and inheritance visibility
- One last discussion

Inheritance

- The three main characteristic/functionalities of OOP:
 - Encapsulation: packaging + data hiding.
 - **Inheritance**: today's topic.
 - Polymorphism: next lecture's topic.
- Through inheritance, we may **create new classes from existing classes**.
 - A **derived (child)** class inherits a **base (parent)** class.
 - A child class has (some) members defined in the parent class.
- This is particularly useful when “**XXX is a OOO**”.
 - An apple is a fruit.
 - A circle is a shape.
 - A truck is a vehicle.

The first example

- Recall that we have defined **MyVector**.
- A two-dimensional (2D) vector is a vector!
- Let's create a class for 2D vector by inheritance.



```
class MyVector
{
protected: // to be explained
    int n;
    double* m;
public:
    MyVector();
    MyVector(int n, double m[]);
    MyVector(const MyVector& v);
    ~MyVector()
    void print() const;
    // =, !=, <, [], =, +=
};
```

Child class MyVector2D

```
class MyVector2D : public MyVector
{
public:
    MyVector2D ();
    MyVector2D (double m[]);
};
MyVector2D::MyVector2D ()
{
    this->n = 2;
}
MyVector2D::MyVector2D (double m[]) : MyVector (2, m)
{
}
```

```
int main()
{
    double i[2] = {1, 2};
    MyVector2D v(i);
    v.print();
    cout << v[1] << endl;

    return 0;
}
```

- Let's all for **MyVector2D**!
 - The modifier **public** will be discussed later.

Inheriting parent class' members

- Members in the parent class are **automatically** defined in the child class.
 - **Except** private members, constructors, and the destructor.
 - A **protected** member can only be accessed by itself and its **successors**.
- What are the members of **MyVector2D**?

```
class MyVector2D : public MyVector
{
public:
    MyVector2D();
    MyVector2D(double m[]);
};
```

```
class MyVector
{
protected:
    int n;
    double* m;
public:
    MyVector();
    MyVector(int n, double m[]);
    MyVector(const MyVector& v);
    ~MyVector();
    void print() const;
    // =, !=, <, [], =, +=
};
```

Invoking parent class' constructors

- The parent class' constructor will not be inherited.
- One of them will be invoked **before** the child class' constructor is invoked.
 - Create the parent before creating the child!
- If not specified, the parent's **default** constructor will be invoked.

```
MyVector::MyVector() : n(0), m(NULL)
{
}

MyVector2D::MyVector2D()
{
    this->n = 2;
    // this->m = NULL is redundant
}
```

```
int main()
{
    MyVector2D v;

    return 0;
}
```

Invoking parent class' constructors

- To **specify** a parent's constructor to call, use the syntax for member initializer:
 - **Pass appropriate arguments** to control the behavior.

```
MyVector::MyVector(int n, double m[])
{
    this->n = n;
    this->m = new double[n];
    for(int i = 0; i < n; i++)
        this->m[i] = m[i];
}
MyVector2D::MyVector2D(double m[]) : MyVector(2, m)
{
    // not MyVector(2, m) here!
}
```

```
int main()
{
    double i[2] = {1, 2};
    MyVector2D v(i);
    v.print();
    cout << v[1] << endl;

    return 0;
}
```


Invoking copy constructors

- How about the copy constructor?
- If we do not define one for the child, the system provides a **default** one.
- **Before** the child's default copy constructor is invoked, the parent's copy constructor will be **automatically** invoked.

```
MyVector::MyVector(const MyVector& v)
{
    this->n = v.n;
    this->m = new double[n];
    for(int i = 0; i < n; i++)
        this->m[i] = v.m[i];
}
class MyVector2D : public MyVector
{
public:
    MyVector2D();
    MyVector2D(double m[]);
    // no copy constructor
};
```

Invoking copy constructors

- If we define a copy constructor for the child, we must **specify** the constructor we want to invoke!
 - Otherwise the parent's **default** constructor will be invoked.

```
class MyVector2D : public MyVector
{
public:
    MyVector2D ();
    MyVector2D (double m[]);
    MyVector2D (const MyVector2D& v) {}
};
```

```
int main()
{
    double i[2] = {1, 2};
    MyVector2D v(i);
    MyVector2D w(v);
    w.print(); // error
    cout << w[1] << endl;

    return 0;
}
```

Using parent's member functions

- Once member variables are set properly, typically all the member functions of the parent can be used with no error.

```
void MyVector::print() const
{
    cout << "(";
    for(int i = 0; i < n - 1; i++)
        cout << m[i] << ", ";
    cout << m[n-1] << ")\n";
}
double& MyVector::operator[] (int i)
{
    if(i < 0 || i >= n)
        exit(1);
    return m[i];
}
```

```
int main()
{
    double i[2] = {1, 2};
    MyVector2D v(i);
    v.print();
    cout << v[1] << endl;

    return 0;
}
```

Invoking parent class' destructor

- When an object of the child class is to be destroyed:
 - First the child's destructor is invoked.
 - **Then** the parent's destructor is invoked **automatically**, even if we do not define a destructor for the child.
- Do not **delete a point twice!**
 - This results in a run time error.

```
MyVector::~~MyVector()  
{  
    delete [] m;  
}  
class MyVector2D : public MyVector  
{  
public:  
    MyVector2D();  
    MyVector2D(double m[]);  
    // no destructor  
};
```

Summary

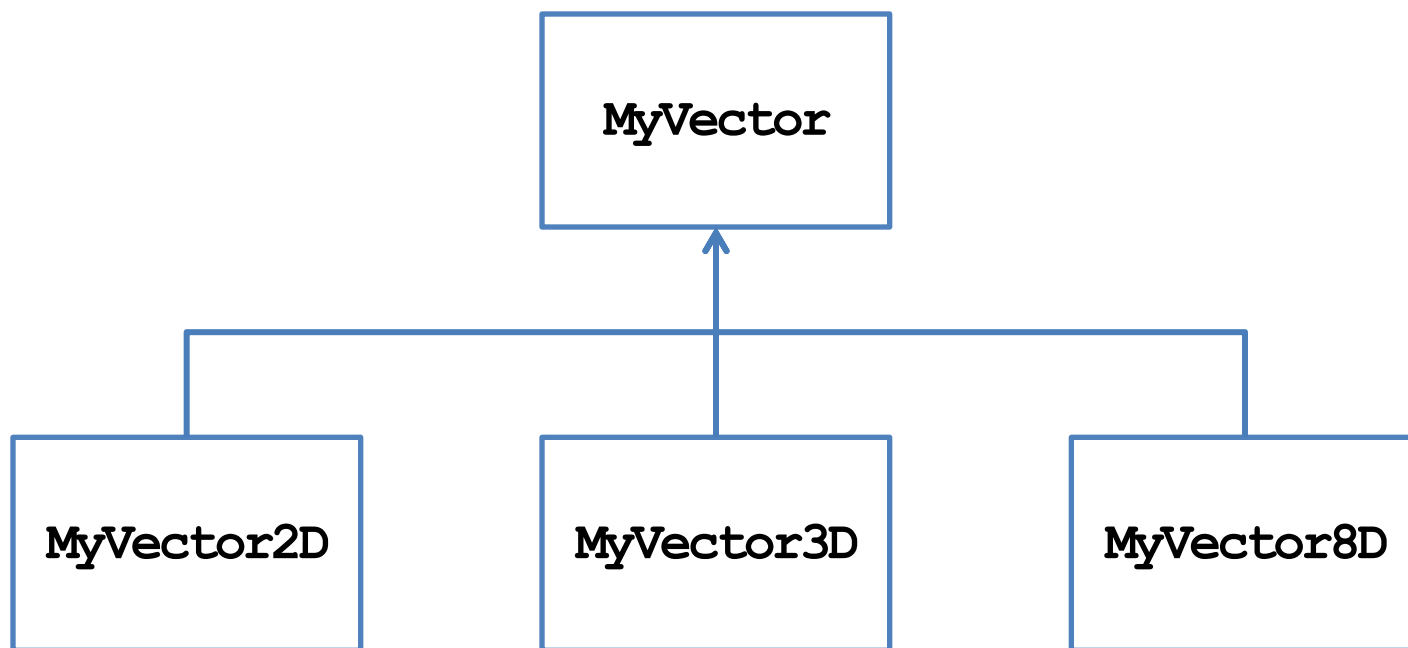
- Using inheritance to create new classes is so simple!

```
class MyVector2D : public MyVector
{
public:
    MyVector2D() { this->n = 2; }
    MyVector2D(double m[]) : MyVector(2, m) {}
};
```

- We save time and enhance **consistency**.
- Pay attention to default constructors, copy constructors, and destructors.
- If one thing should not be inherited, set it to private.

Brothers and sisters

- One parent class can be inherited by multiple child classes.



Outline

- Basic ideas and the first example
- **Defining new members and overriding old members**
- Cascade inheritance and inheritance visibility
- One last discussion

Defining new members for the child

- A child may have **its own members**.
 - The parent has no way to access a child's member.
- Let's define a **setValue()** function without using arrays:
 - Note that this should never be a member of **MyVector**.
- We may also define new member variables and static members.

```
class MyVector2D : public MyVector
{
public:
    MyVector2D() { this->n = 2; }
    MyVector2D(double m[]) : MyVector(2, m) {}
    void setValue(double i1, double i2);
};
void MyVector2D::setValue(double i1, double i2)
{
    if(this->m == NULL)
        this->m = new double[2];
    this->m[0] = i1;
    this->m[1] = i2;
}
```


Function overriding

- We may also redefine existing member inherited from a parent.
 - This typically happens to member functions.
 - We say that we **override** the member function.
- As an example, let's override **print()**:

```
class MyVector2D : public MyVector
{
public:
    MyVector2D() { this->n = 2; }
    MyVector2D(double m[]) : MyVector(2, m) {}
    void setValue(double i1, double i2);
    void print() const;
};

void MyVector2D::print() const
{
    cout << "2D: (";
    for(int i = 0; i < n - 1; i++)
        cout << m[i] << ", ";
    cout << m[n-1] << ")\n";
}
```

Function overriding

- To override a parent's member function, define a child's member function with exactly the same **function signature**.
 - A child object will invoke the child's implementation.
 - The parent's implementation becomes hidden to a child object.
- Inside the child class, we may invoke a parent's member function by using `::`.

```
void MyVector2D::print() const
{
    cout << "2D: ";
    MyVector::print();
}
```

- Use it if consistency can be enhanced.

Overriding a constant function

- What will happen to the following program?

```
int main()
{
    double i[3] = {1, 2};
    const MyVector2D v(i);
    v.print(); // ?

    MyVector2D u;
    u.setValue(3, 4);
    u.print(); // ?

    return 0;
}
```

```
class MyVector
{
    // ...
    void print() const;
};

class MyVector2D : public MyVector
{
    // ...
    void print() { MyVector::print(); }
    void print() const
    {
        cout << "2D: ";
        MyVector::print();
    }
};
```

Overriding a constant function

- How about this?

```
int main()
{
    double i[3] = {1, 2};
    const MyVector2D v(i);
    v.print(); // error!

    MyVector2D u;
    u.setValue(3, 4);
    u.print(); // ?

    return 0;
}
```

```
class MyVector
{
    // ...
    void print() const;
};

class MyVector2D : public MyVector
{
    // ...
    void print() { MyVector::print(); }
};
```

Overriding a member variable?

- Technically, we may override a member variable.
- For example, it seems to be a good idea to override `n` by a **static constant**:

```
class MyVector2D : public MyVector
{
private:
    static const int n;
public:
    MyVector2D() { // no this->n = 2; }
    MyVector2D(double m[]) : MyVector(2, m) {}
    // not overriding print();
    void setValue(double i1, double i2);
};

const int MyVector2D::n = 2;
```

```
int main()
{
    double i[3] = {1, 2};
    const MyVector2D v(i);
    v.print(); // good :)

    MyVector2D u;
    u.setValue(3, 4);
    u.print(); // bad :(

    return 0;
}
```

Overriding a member variable?

- As **print()** is not overridden, **v** and **u** invoke the parent's implementation.
- The parent's implementation uses **MyVector::n**.
 - With the constructor with an array as a parameter, **MyVector::n** is set to 2.
 - With the default constructor, **MyVector::n** is set to 0!
- Even though **MyVector2D::n** is set to 2, it is not used in **MyVector::print()**.
- What if we also override **print()**?

```
void MyVector2D::print() const
{
    cout << "2D: (";
    for(int i = 0; i < n - 1; i++)
        cout << m[i] << ", ";
    cout << m[n-1] << ") \n"; // good :)
}
```

```
void MyVector2D::print() const
{
    cout << "2D: ";
    MyVector::print(); // bad :(
}
```

Overriding parent's member

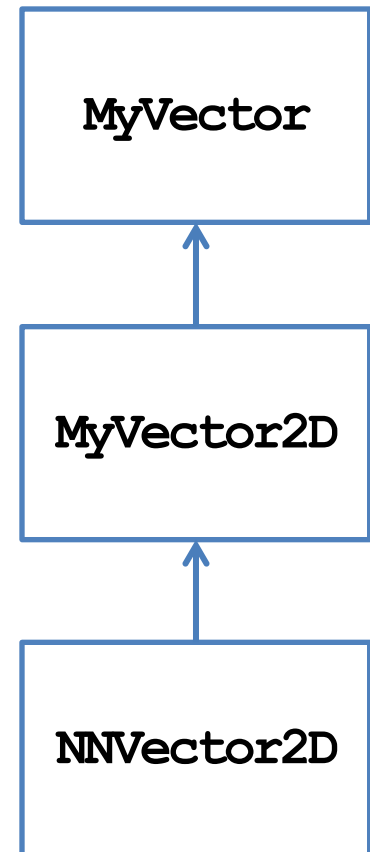
- In general, overriding a parent's member variable is not suggested.
 - Unless you really know what you are doing.
 - After all, we will inheritance because we believe XXX is a OOO. A parent's member variable should be a part of a child!
- Overriding a parent's member function is useful.
- What is the difference between function overloading and function overriding?
- Sometimes we override a member function for efficiency.

Outline

- Basic ideas and the first example
- Defining new members and overriding old members
- **Cascade inheritance and inheritance visibility**
- One last discussion

Cascade inheritance

- While a child inherits its parent, it may have a grandchild inheriting itself.
- How may we create a class for two-dimensional nonnegative vectors?
 - $\{(x, y) \mid x \geq 0, y \geq 0\}$.
- A 2D nonnegative vector **is a** 2D vector!
- Let's use inheritance again.



Child class `NNVector2D`

- Defining `NNVector2D` is simple:

```
class NNVector2D : public MyVector2D
{
public:
    NNVector2D(); // do we need it?
    NNVector2D(double m[]);
    void setValue(double i1, double i2);
};
NNVector2D::NNVector2D()
{
}
```

```
NNVector2D::NNVector2D(double m[])
{
    this->m = new double[2];
    this->m[0] = m[0] >= 0 ? m[0] : 0;
    this->m[1] = m[1] >= 0 ? m[1] : 0;
}
void NNVector2D::setValue
(double i1, double i2)
{
    if(this->m == NULL)
        this->m = new double[2];
    this->m[0] = i1 >= 0 ? i1 : 0;
    this->m[1] = i2 >= 0 ? i2 : 0;
}
```

- Why not specifying a parent's constructor?
- What happens when an `NNVector2D` object is created?

Child class NNVector2D

- How about

```
NNVector2D::NNVector2D(double m[]) : MyVector2D(2, m)
{
    if(m[0] < 0)
        this->m[0] = 0;
    if(m[1] < 0)
        this->m[1] = 0;
}
```

or

```
NNVector2D::NNVector2D(double m[])
{
    setValue(m[0], m[1]);
}
```

Cascade inheritance

- In general, a class has all the protected and public members (excluding constructors and destructors) of its **predecessors**.
- When an object is created:
 - Constructors are invoked from the oldest class to the youngest class.
 - Each constructor can specify a **one-level-above** constructor to invoke.
 - Only one level!
- When an object is destroyed:
 - Destructors are invoked from the youngest to the oldest.

Inheritance visibility

- Recall that we added the modifier **public** when **MyVector2D** inherits **MyVector** and when **NNVector2D** inherits **MyVector2D**.
 - This modifier specifies the **inheritance visibility**.
 - It shows how this child modify the member visibility set by its predecessors.
- When one inherits something from its parent, it may **narrow** the **visibility** of these members.
 - E.g., if my parent set it to protected, I may set it to private.
 - E.g., if my parent set it to private, I cannot set it to public.
- Why only narrowing?

Inheritance visibility

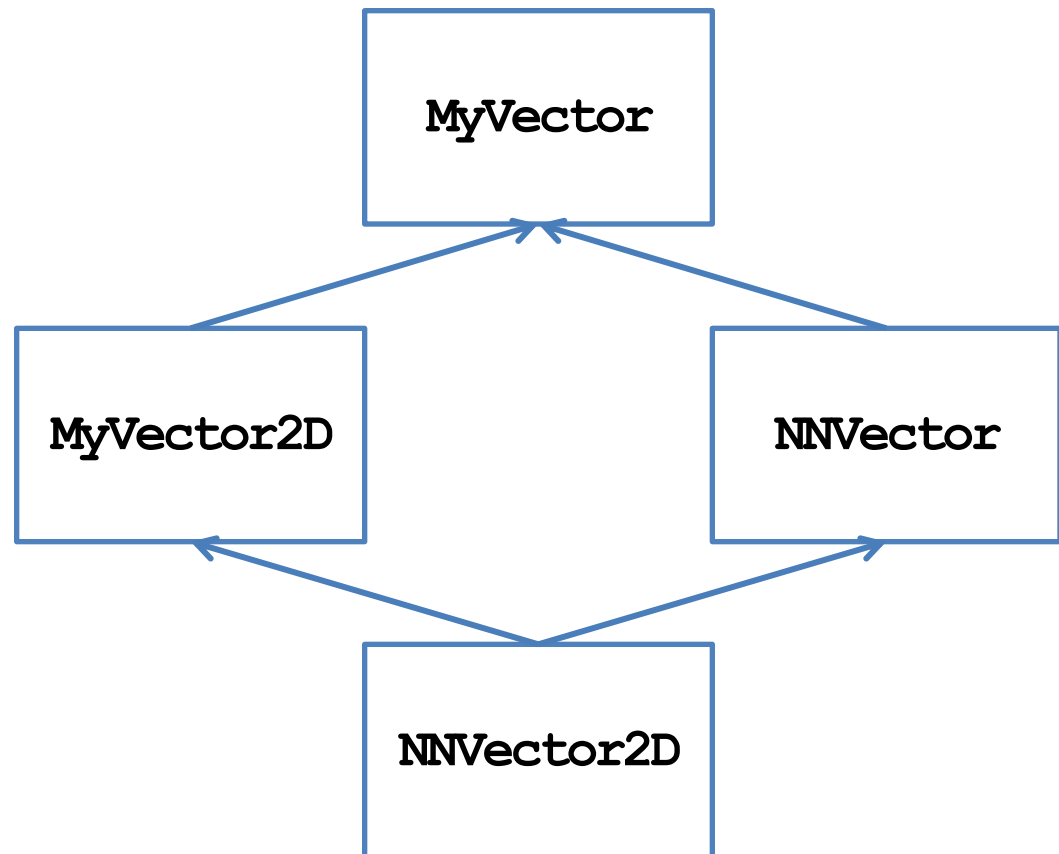
- In general, the visibility of a member in a child class depends on:
 - The member visibility by the parent.
 - The inheritance modifier.

Member visibility by the parent	Inheritance modifier		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

- If you have no idea, just use public inheritance.

Multiple inheritance

- Suppose your friend argues:
 - A two-dimensional vector is a vector.
 - A nonnegative vector is a vector.
 - A two-dimensional nonnegative vector should be the child of them!
- Does that make sense?



Multiple inheritance

- In C++, **multiple inheritance** is allowed.
- However, it is not recommended!
 - In **some other** object-oriented programming languages (e.g., Java), multiple inheritance is forbidden.
- If there are multiple parents:
 - Whose constructor/destructor goes first?
 - Whose variables are stored in the front?
 - May I inherit from my sister? May I inherit from my grandaunt?
- We also suggest you not to do multiple inheritance (even though it has been used in C++ standard library).

Outline

- Basic ideas and the first example
- Defining new members and overriding old members
- Cascade inheritance and inheritance visibility
- **One last discussion**

A problem regarding the overridden =

- In fact, the definition of MyVector2D contains a flaw.
 - It is possible to have an object with **n = 2** but **m = NULL**.
- This is inconsistent.
- How to fix it?

```
int main()
{
    double i[3] = {1, 2};
    NNVector2D v(i);

    NNVector2D u;
    u = v; // error
    u.print();
    return 0;
}
```

```
const MyVector& MyVector::operator=
(const MyVector& v)
{
    if(this != &v)
    {
        if(this->n != v.n)
        {
            delete [] this->m;
            this->n = v.n;
            this->m = new double[this->n];
        }
        for(int i = 0; i < n; i++)
            this->m[i] = v.m[i];
    }
    return *this;
}
```

A question regarding the overridden =

- Beside the previous issue, is there anything weird with the overridden =?

```
const MyVector& operator=(const MyVector& v) ;
```

- The type of the parameter is **MyVector**, not **MyVector2D**!
 - Has **MyVector2D** been casted to **MyVector**?
 - May we cast **MyVector** to **MyVector2D**?
 - Is casting between classes allowed if they are not parent and child?
 - What if they are brothers or sisters (like **MyVector2D** and **NNVector**)?
 - How may we utilize casting between classes?
- We will discuss these questions when we introduce **polymorphism**.