Object Oriented Programming with C++

2024 Spring Semester

21 CST H3Art

Chapter 8 Inheritance: Extending Classes

- **Inheritance（继承） a.k.a. derivation（派生）**
  - Mechanism of creating a new class from an old one. The new class **inherits all of the traits** from the existing one.
  - The old class: **base class（基类）** or **parent class（父类）**
  - The new one: **derived class（派生类）** or **subclass （子类）**
  - Inheritance is able to define new classes of objects using existing classes as a base
    - The new class inherits **the attributes（属性，即数据成员）** and **behaviors（行为，即方法/函数）** of the parent classes
    - New class is a specialized version of the parent class
- **Reusability（可重用性）**
  - Save time and money
  - Reduce frustration
  - Increase reliability
- **Class Derivation（派生类）**
  - Syntax:

    ```
    class derived_class_name: visibility_mode base_class_name{
        ...
    };
    ```

    - **visibility_mode（可见模式）**: specifies access to the base class members, `public`, `protected`, `private`, and `private` **by default**
  - For `public` derivation `class C: public A`:
    - The inherited **public** members of `A` appear **as public** members of `C`
    - The inherited **protect** members of `A` appear **as protect** members of `C`
    - The inherited **private and unaccessible** members of `A` appear **as unaccessible** to `C`
  - For `private` derivation `class C: private B`:
    - The inherited **public** members of `B` appear **as private** members of `C`
    - The inherited **protect** members of `B` appear **as private** members of `C`
    - The inherited **private and unaccessible** members of `B` appear **as unaccessible** to `C`
  - For `protected` derivation `class C: protected B`:
    - The inherited **public** members of `B` appear **as protect** members of `C`
    - The inherited **protect** members of `B` appear **as protect** members of `C`
    - The inherited **private and unaccessible** members of `B` appear **as unaccessible** to `C`
  - **Private vs. Protected Members（比较继承的私有和保护成员）**
    - Inherited **private** members **CANNOT** be **accessed directly by name** in derived class
    - Inherited **protected** members **can** be **accessed directly by name** in derived class
- **Not "All" Members Inherited（并非所有成员都会被继承）**
  - **Base class members（基本类成员）** not inherited in derived class:
    - Constructors
    - Destructor
    - Copy constructor

    - Assignment operator
- Sometimes need to be invoked in derived class **except for destructors（除了析构函数外有时需要在派生类内调用父类的基本类成员）**
    - Destructors are always automatically invoked
- **Base class constructors（基类构造函数）** are **NOT inherited** in derived classes! But they can be invoked within derived class constructor
- Base class constructor initialize base class member variables
    - **Default constructor** of direct base class is called automatically by derived class constructor
    - Parameter constructors of direct base class are called by including them in the **initializer list（在初始化列表中直接调用基类的构造函数）**
    - **"First" thing derived class constructor** does
    - Derived class constructor can **not** call **indirect base class constructor（不能调用非直接继承类的构造函数）**
- Example:

```cpp
#include <iostream>

using namespace std;

class A {
  int a;

public:
  A() { cout << "A" << endl; }
  A(int sa) {
    a = sa;
    cout << "A" << endl;
  }
  ~A() { cout << "~A" << endl; }
};

class B : public A {
  int b;

public:
  B(int sa, int sb) : A(sa) {
    b = sb;
    cout << "B" << endl;
  }
  ~B() { cout << "~B" << endl; }
};

class C : public B {
  int c;

public:
  C(int sa, int sb, int sc) : B(sa, sb) {
    c = sc;
    cout << "C" << endl;
  }

  // Error, cannot invoke the indirect base class
  // C(int sa, int sb, int sc) : A(sa), B(sb) {
  //   c = sc;
  //   cout << "C" << endl;
  // }

  ~C() { cout << "~C" << endl; }
};

int main() {
  C objc(1, 2, 3);
  return 0;
}
```

Output:

```
A
B
C
~C
~B
~A
```

  - When calling multiple constructors in initializer list, the calling order **depends on the member declare order（初始化取决于类成员定义的顺序而不是初始化列表的顺序）** instead of initializer list order.
- **Constructor: No Base Class Call（不调用基类构造函数）**
  - Derived class constructor should **always invoke one of the base class's constructors（总是需要在派生类的构造函数中调用基类的构造函数）**
  - If you do not, then the **default base class constructor automatically called（如果不手动调用基类的构造函数，基类的默认构造函数会被调用）**
- **The "Big Three"（三大件）**
  - **Destructors（析构函数）**
    - Only needed when pointers and **dynamic memory allocation** are used
  - **Copy constructors（拷贝构造函数）**
    - NOT inherited, but can be used in derived class definitions, similar to how derived class constructor invokes base class constructor **（不被直接继承，但可以在派生类的定义中被使用，参见在派生类中调用基类构造函数的方式）**
    - Example:

```
Derived::Derived(const Derived& Object): Base(Object), ... {
  ...
}
```

  - **Assignment operators（赋值运算符）**
    - NOT inherited, same as copy constructor
    - Example:

```
Derived& Derived::operator=(const Derived & rightSide) {
  if (this != &rightSide) //avoid self assignment
    Base::operator=(rightSide); // 先调用基类的赋值运算符
  ... // 接下来再编写派生类的赋值部分
}
```

- **Redefinition（重定义）**
  - A derived class can redefine members defined in its parent class. With redefining:
    - the method in the child class **has the identical signature（拥有相同的函数签名：函数的名称、参数类型和参数个数的组合形式，不包括函数返回类型）** to the method in the base class
    - a child class **implements its own version** of a base class method
  - Example:

```cpp
#include <iostream>

using namespace std;

class Point {
  int x, y;

public:
  void set(int a, int b) {
    x = a;
    y = b;
  }
  void print(){
    cout << "Point: x = " << x << ", y = " << y << endl;
  }
};

class Circle : public Point {
  double r;

public:
  void set(int a, int b, double c) { // Overload
    Point::set(a, b);
    r = c;
  }
  void print(){ // Redefinition
    cout << "Circle: radius = " << r << endl;
  }
};
```

- **Redefining vs. Overloading（比较重定义与重载）**
  - **Redefining（重定义）** in derived class:
    - **SAME** parameter list **（同样的参数列表）**
    - Essentially **"re-writes" same function（重写了相同的函数）**
  - **Overloading（重载）**:
    - Different parameter list **（不同参数列表）**
    - Defined "new" function that takes different parameters
    - Overloaded functions must **have different signatures**