Summary for CSC3002 Part2

OrganicFish

November 10, 2022

Contents

1 Introduction to Class **4**

1.1 Classification 4

1.2 Enumerated Types 4

1.2.1 C++ enum 4

1.2.2 C-style enum 4

1.3 Structures 5

1.4 Class 5

1.4.1 Constructors 5

1.4.2 Parsing 7

1.4.3 Dealing with pointer fields 7

1.4.4 Copy 7

1.4.5 Move 8

1.4.6 Other interesting methods 9

2 Class Inheritance **10**

2.1 Syntax 10

2.2 Access modifiers 10

2.2.1 Accessibility 10

2.2.2 friend 10

2.3 Virtual Method 11

2.3.1 Standard syntax 11

2.3.2 Implementation 11

2.3.3 Pure virtual method 11

2.3.4 Override 12

2.3.5 Private + virtual, and final keyword 12
2.3.6 Virtual destructors and constructors 13
2.3.7 Pointers and Virtual table 13
2.4 The order of constructor calling 14
2.5 Slicing and Casting 15
2.6 Inheritance Relationship 15
2.6.1 ”L” in SOLID - Square is NOT a rectangle 15
2.6.2 Multiple Inheritance - Deadly diamond 15
2.6.3 Virtual functions with default parameters 16

To my dear assisstant, Christina.

1 Introduction to Class

1.1 Classification

Three sorts of data types in C++:

1. Primitive Types: int, double, bool, char...
2. Collections, Containers.
3. Compound data types: Enumerated types, structures, and classes.

1.2 Enumerated Types

1.2.1 C++ enum

1. Syntax:

```
    // Declaration
    enum class Direction {NORTH, EAST, SOUTH, WEST};

    // Instantiation
    Direction myDirect = Direction::EAST;

    // Output
    cout << int(myDirect) << endl;  // 1
```

Remark1: NORTH is automatically set to 0.

Remark2: Direction:: is compulsory to access to the values inside the enum collection.

Remark3: Enum is just a collection of integers with aliases. But generally, you can only use those aliases to pass value. e.g.,

```
    myDirect = 2; // Not allowed!
    myDirect = (Direction)2;
    myDirect = Direction::WEST;
```

1.2.2 C-style enum

```cpp
    // Declaration
    enum Direction {NORTH,SOUTH,...};
```

Remark: The names inside the enum is GLOBAL.

1.3 Structures

Omitted. This type has already been supplanted by Class.

1.4 Class

1.4.1 Constructors

Example:

```cpp
class Cell {

  public:
    // constructor
    Cell() = default;

    // normal
    Cell(int sz): m_sz(sz) {
      create(sz);
    }

    // copy constructor (delegate to normal constructor)
    Cell(const Cell & src) {
      copy(src);
    }

    // copy assignment
    // Usage: c2 = c1;
    Cell & operator=(const Cell & rhs) {
      Cell tmp {rhs}; // call copy constructor
      swapFields(tmp);
      return *this;
    }

    // move constructor
    Cell(Cell && src) {
      swapFields(src);
    }

    // move assignment
    Cell & operator=(Cell && rhs) {
      swapFields(rhs);
      return *this;
    }
};
```

0. Note that Cell c = a is equivalent to Cell c(a). Examples: Cell c1(1) and Cell c1 = 1 (Yes, this one is Valid!!!); For simplicity, this note will take the form of c1(1).

1. Normal: Cell c1(2); Cell c1(Cell(12)); Cell c1(createCell());.

Remark: The last two are because of copy elision: That happens in the initialization of an object, when the source object is a nameless temporary and is of the same class type as the target object. When the nameless temporary is the operand of a return statement, this variant of copy elision is known as RVO, "return value optimization". (As it is in createCell()).

2. Copy: Cell c2(c1); Cell c2(c1);.

3. Copy Assignment: c2(c1);.

4. Move:

```
    vector<Cell> v;
    v.push_back(Cell(1)); // Method 1
    Cell c2(move(c1)); // Method 2
```

5. Move Assignment: c2(move(c1));.

Remark: Assignment or Construction, depends on whether you're creating a new instance.

1.4.2 Parsing

A very confusing syntax:

```
Cell c1(Cell(1));
```

It is not creating a cell object! It is defining a function instead.

You may want this:

```
Cell c1{Cell(1)};
```

1.4.3 Dealing with pointer fields

When the class incorporates any field which is a pointer, the constructor should use NEW, and the destructor should correspondingly use DELETE. Otherwise, the constructor allocate a new space and let the pointer point to it. But the new space in memeory will be freed after construction.

The following codes will cause such a problem.

```
Cell c1(int sz) {
    int temp[sz]{0};
    this->arr = temp;
}
```

1.4.4 Copy

1. The problem of shallow copy

   By default, C++ explicitly defines the copy constructor using shallow copy.

```
// Field:
int* p;
// Copy Constructor
Cell (Cell & c1) {this->p = c1.p;}
```

Double freeing memory:

Destructor of c1: release the space pointed by p;

Destructor of the new Cell obj: release the same space.

2. Deep copy

If there is a pointer member in the class, we should manually define the copy constructor/assignment using deep copy.

That is, to new a block of memory, copy the space pointed by c1.p into it, and let the new pointer point to this space.

Related destructor:

1. Check whether p is a nullptr;
2. If not, delete p;
3. Then, set p to be a nullpty.

Related copy constructor: e.g., c1 = c2.

1. Free c1.p;
2. New, copy, and let c1.p point.

OR, we have another approach:

1. Create a temporary object based on c2.
2. Swap the fields of c2 and the temp object.
3. Temp object is automatically released.

1.4.5 Move

1. Why we need rvalues and move?

1.1 How to avoid copy.

Case 1: No reference + deep copy: Copy twice? Copy infinitely!

```
  // Error
  Cell(Cell c) {"Deep Copy"} // Deep copy infinitely! (Parameter itself
 calls the copy constructor)
  Cell c1(c2);
```

Case 2: Reference + deep copy: Copy once

```
  Cell c1(const Cell & c) {"Deep Copy"}
```

Case 3: Reference + copy by moving?

```
Cell c1(const Cell & c) {
    this->arr = c.arr;
    // To avoid double freeing... But error because of const reference.
    c.arr = nullptr;
}
```

Final solution: move

```
Cell c1(const Cell && c) {
    this->arr = c.arr;
    c.arr = nullptr;
}
```

2. Real example

```
// std::vector
void push_back(const value_type& val);
void push_back(value_type&& val); // This leads to the discarding of the
rvalue.
```

1.4.6 Other interesting methods

1. Initializer List:

```
Cell(int size) : sz(size) {}
Cell(const Cell & c) : Cell(c.sz) {copy(c);}
```

2. Constant member function.

```
int get_size() const {return this->sz;}
```

The function is not allowed to modify any member. We may understand this by considering the hidden this:

```
int get(const Cell* this) {...}
```

They are basically the same. Likewise,

```
int get() const&& {return sz;}
// (Perhaps) This is equivalent to:
int get(const Cell* && this) {return this->sz;}
// Usage:
Cell(10).get();
```

2 Class Inheritance

2.1 Syntax

```
class Base {};
class Derived : public Base {};
```

2.2 Access modifiers

2.2.1 Accessibility

1. Private members cannot be inherited.
    2. Using public to inherit: public - public, protected - protected.
    3. Using protected/private: public and protected - protected/private.

2.2.2 friend

1. Friend function

    Public func:

    friend void func();

    Methods of other classes:

```cpp
friend void Class_name::func();
```

Remark: Cannot declare the private methods of other classes to be friend.

2. Friend class

```cpp
friend void Class_name;
```

Remark: Friend can be declared anywhere inside the class.

2.3 Virtual Method

2.3.1 Standard syntax

```cpp
class Base {
    virtual void show();
};
class Derived : public Base{
    virtual void show() {...}; // This virtual keyword is optional
};
```

2.3.2 Implementation

1. A virtual method must be resolved somewhere along any path of the inheriting chain.

2. If a virtual method is not resolved before and in some derived class, then you cannot create an object of this derived class.

2.3.3 Pure virtual method

0. Syntax: virtual void func() = 0;

1. The class containing a pure virtual method is an abstract class.

2. If the derived class of an abstract class doesn't resolve the pure virtual method, then it is still an abstract class.

3. You cannot create any object of the abstract class.

2.3.4 Override

1. If you are resolving a virtual function in the derived class, you may miss part of the signature of the virtual method in the base class, thus ”overriding” wrong virtual method.

To avoid this, add an override keyword when ”overriding” a virtual method. When there is no matching method with the same signature, you will receive an error!

Preferable syntax:

```cpp
class Derived : public Base{
    void func() override {};
    // Instead of...
    virtual void func() {};
};
```

2. Override will ”hide” other functions.

(Clarification of the slide from Week 6.)

This has nothing to do with the ”override” keyword.

When the base class has several virtual functions with the same function name but different signatures, and you are overriding (with the override keyword or not)only one of them, then, other functions inside this derived class, are hidden. That is, though they are inherited, you cannot invoke them using the pointer to an derived object.

2.3.5 Private + virtual, and final keyword

1. Override is regardless of accessibility. This is related to vtable.

2. Which virtual method to invoke depends on the real object, beacuse it stores the vptr.

3. However, accessibility is related to the type of the pointer.

```cpp
class Base {
public:
    virtual void func();
};
class Derived : public Base {
private:
    void func() override {"In Derived"};
};
```

```

    Base *p = new Derived();
    p->func(); // In derived
    Derived *p1 = new Derived();
    p1->func(); // Error!
```

4. Use final to stop inheriting/overriding

```
    virtual void someMethod() final {};
    class Super final {};
```

2.3.6 Virtual destructors and constructors

1. Destructors should be virtual whenever possible.
2. Constructors can never be virtual.

2.3.7 Pointers and Virtual table

1. What if I don't use virtual?

```
    class Base{
    public:
        void func() {"In Base"}
    };
    class Derived : public Base {
        void func() {"In Derived"}
    };
    // Testing...
    Base *p = new Derived();
    p->func(); // Base
```

Remark: If func() is virtual, then the result will be "In Derived".

2. Base pointer to a derived object.

```
In base: virtual vfunc(){"In Base"}; doVfunc(){vfunc();}
In derived: vfunc() override {"In Derived"};
    Base *p1 = new Base();
    Base *p2 = new Derived();
    p1->doVfunc(); // Base
    p2->doVfunc(); // Derived
```

3. Static and Dynamic binding

Static binding: Knowing which function to invoke when compiling.

Dynamic binding: Not knowing which to invoke until executing.

4. Vtable and vptr

Whenever a virtual func is declared inside a class, there is a virtual table created for the class. It consists of addresses to the vfunctions of the class.

The object of such a class contains a virtual pointer pointing to (the base address of) the vtable.

Thus, when a pointer tries to access to a virtual function, it actually refers to the vptr of the real object, eventually calling the overriden version of that function.

2.4 The order of constructor calling

```
In Base: Base() {"1"}; virtual ~Base() {"1"};
In someClass: someClass() {"2"}; virtual ~someClass() {"2"};

    class Derived : public Base {
    public:
        Derived() {"3"};
        virtual ~Derived() {"3"};
    protected:
        someClass c;
    };
    Base *p = new Derived();
    delete p;
    // Output: 123321
```

General Rule:

Order of constructors: Base, member, derived.

Order of destructors: Derived, member, base.

Which parent constructor: By default, call Base().

However, you can specify which to call in the initializer list. Even in such a case, the base constructor still precedes the member constructor, which precedes the derived constructor.

2.5 Slicing and Casting

See the section pointers and vtable.

2.6 Inheritance Relationship

2.6.1 ”L” in SOLID - Square is NOT a rectangle

Liskov Substitution principle: Generally objects of a superclass should be replaceable with objects of its subclasses without breaking the application.

Let’s say the rectangle class has methods to set length and width independently. It’s clear that you cannot set these properties of a square seperately, so the principle is broken.

Generally speaking, we should let the subclass incorporate all the abilities of its superclass.

2.6.2 Multiple Inheritance - Deadly diamond

1. You should not directly create a diamond relationship. This will cause an error when you are trying to use any member from class A(the very first base class).

2. You may fix this by specifying which member you are using:

e.g., B::a or C::a.

3. You may fix this by using virtual inheritance.

```
  class Derived : virtual public B, virtual public C {};
```

With this, the father constructor is called only once.

However, if both B and C redefines a in A, the ambiguity still emerges...

Turn to the previous solution to fix this...

2.6.3 Virtual functions with default parameters

In short: The function can be overriden, but the default parameters are NOT inherited...

```
    virtual void func(int i = 1) {"In base: i"};
    void func(int i = 2) override {"In derived: i"};
    // The most confusing part happens when...
    Base *p = new Derived();
    p->func(); // In derived: 1
```