COMP2012: Data Structure

Hong, Lanxuan

Spring 2025

This is the note for COMP 2012: OOP & Data Structures.

Last updated on Tuesday 16th September, 2025.

Contents

1 Review on C++

1.1 Pointer & Reference

```
int var = 10;
int* ptr = &var; // & returns memory address, int* stores this address
*ptr = 5; // * access the value stored in address
```

Note ⚠ `*` and `&` are **unary operands**, i.e. they only have single operand. e.g. for `int* a,b,c`, only a is a pointer variable.

1.2 Const Keyword

Const-ness can be applied on:

- constant variable(read-only) and constant pointer(cannot change binding).
- const reference as arguments: protect variables from being modified.
- constant member functions: cannot modify data members.
- **constant object**: put `const` keyword before constructor to create const object. For a const object, only constant member functions can be called so that data members cannot be modified.

1.3 New C++ 11 Features

1.4 Separate Compilation and Make-file

2 Constructor & Destructor

2.1 Class Object Initialization

There are 4 types of constructors in terms of function:

- Default Constructor. e.g. `Word movie;` **called when object is created**
- Conversion Constructor: accept only **1** argument and specifies a conversion. . e.g. `Word movie("Avatar")` or `Word movie"Avatar"` or `Word movie = "Avatar"(Implicit, more expensive)`
- Coly Constructor: accept only **1** const reference of the object of the same class type. C++ will do member wise copy by default. **called when pbv or rbv**
- General Constructor. e.g. `Word* p = new Word("action",1)`

Note ⚠ C++11 allows default values for non-static data members of a class.

```
class Word{
    private:
        int frequency {0}; // default initializer
        const char* str {nullptr};
    public:
        Word(){}; // default constructor
        Word(char* s) {// conversion construtor
            str = new char[strlen(s)+1];strcpy(str,s);}
        explicit Word(char c){ str = new char[2]; str[0] = c; str[1] = '\0';}
        Word(const Word& w){ //copy constructor
        frequency = w.frequency;
        str= newchar[strlen(w.str)+1]};
        strcpy(str,w.str);}
void print_word(Word x){}
int main(){
    print_word("Avatar"); // work! by implicit conversion
    print_word('A'); // error! implicit conversion disallowed
    Word obj = 'A'; // error! implicit conversion disallowed
};
```

Note ⚠ When there is no user-defined constructors are found, the compiler automatically supply the simple default constructor `Word()`. If any constructor is defined, the object can only be defined by calling a constructor.
⚠ Add `explicit` before constructor to disallow implicit conversion.
⚠ Copy elision and return value optimization
⚠ Double-free **(run-time error)** caused by member-wise copying the pointer

2.2 Member Initializer List(MIL)

MIL after constructor initializes the data members at the time when creating an object even before data member initialization. ⚠ **important when creating constant data members or reference data members!** The content inside {} or () are parameters passed in or just a value, which resolves name conflict. It also specifies the way to construct data member objects.

```
Word(int frequency) : frequency(frequency), str(nullptr){}; // delegated constructor
```

MIL is also powerful for creating **delegating constructor**(C++11 or later).

```
Word(const Word& w): Word(w.frequency){};
```

2.3 Destructor

Note ⚠ Compiler-generated member functions

- default constructor/destructor (if no user-defined constructor)
- default copy constructor (member-wise copy)
- default copy assignment operator function
- `= default` to setup default and `= delete` to forbid default settings

2.4 Construction-Destruction Order

Class A **has** Class B means object of Class B is a data member of class A. Class A **owns** Class B means there is a pointer of Class B in Class A which gives the possibility to create Class B in Class A.

3 Inheritance

3.1 The idea of inheritance

The key idea of inheritance is to find out the common data members and member functions from **derived class**, put them into a **parent class/base class**, and apply the **inheritance** mechanism.

```
#ifndef UPERSON_H
#define UPERSON_H
#include "uperson.h"
class Student: public UPerson // public inheritance
{
    private:
        int num_courses;
    public:
        Student(string n, Department d):
            UPerson(n,d),num_courses(0){ }
};
class Teacher: public UPerson // public inheritance
{
    private:
        int rank;
    public:
        Teacher(string n, Department d):
            UPerson(n,d), rank(0){ }
}
#endif
```

Inheritance implements the **is-a relationship** following **Liskov Substitution Principle** . If class D(derived class) inherits from class B(base class), then every D object is also a B object, but not vice versa.

Note ⚠ Function expecting an argument of type B/pointer to B/B reference will also accept D/pointer to D/D reference.

3.2 Inheritance hierarchy

Note ⚠ For the data members in derived class inherit from the base class, it is not safe to directly initialize them in the derived class. A better solution is to call parent class constructor in MIL.(MIL does not control ordering)

The order of constructing an object of a class follows: its parent, data members, then itself. The order of destruction is exactly the other way around.

There are some issues when applying inheritance hierarchy.

1. **slicing**: An assignment(**copy**) from a derived class object to a base class object result in slicing, which lose the trace of things defined in the derived class.
2. **name conflicts**: follow the idea of scopes.

3.3 Access Control

The access of data-member and member functions of a class is given by access keywords, i.e., **private** and **public**.

Though the derived class grab all data members from base class, the derived class have no access to private data members in its parent class. The **protected** keyword `protected` provide accessibility for not only the member functions and **friends** (**friends function/class of a class can access its private members**) of the class, but the member functions and friends of subclasses, i.e.**all its derived classes**.

Note ⚠ Friend mechanism
this is the text for friend mechanism.
this is the text for friend mechanism.
this is the text for friend mechanism.
this is the text for friend mechanism.

Using protected means less data encapsulation(⚠ **which is bad**) while higher flexibility for derived classes to directly access its members.

3.4 Polymorphism and Dynamic Binding

```
class UPerson{
    void print()const{};}
class Student:public UPerson{
    void print()const{};}
void print_label(const UPerson& uperson){uperson.print();}
Student student;
print_label(student); \\ call print function in UPerson
```

In the above example, the variable student of `student` type can be accepted as `Uperson` argument because of the **polymorphic substitution principle**(Liskov principle). However, when these function codes are compiled, the compiler only looks at the **static type** of the argument `uperson` and the method `UPerson::print()` is called.

By default, C++ use static binding, or early binding(**type check at compilation**). In static binding, what a pointer really points to, or what a reference actually refers to is not considered; only the pointer/reference type is. C++ also allows **dynamic binding** which is supported through **virtual function**. When dynamic binding is used, the method to be called is selected using the **actual type**(**pass by reference or pointer, no slicing**) of the object in the call. If a function is defined as virtual, the type is determined at runtime, i.e.**RTTI**(Run time type information).

```
class UPerson{
    virtual void print()const{};}
class Student:public UPerson{
    virtual void print()const{};}
void print_label(const UPerson& uperson){uperson.print();}
Student student;
print_label(student); \\ call print function in Student
```

Note ⚠ Type Casting

1. static cast: `static_cast` at compile time.
2. dynamic cast: `dynamic_cast` at run time(safer but slower). **pointer/reference cannot be casted from derived type to base type**

In the above example, `Student::print()` **overrides** `UPerson::print()`(**Not Overloading!**). Overriding is impossible if a method is not virtual. Use `override` keyword(C++11 afterwards) to specify and check this function.

The constructors and destructor can be more powerful with virtual function. For example, we can define destructor as virtual.

```
UPerson* ps = new Student;
delete ps; \\ ok if UPerson destructor is virtual, call Student destructor
```

3.5 Abstract Base Class

Note ⚠ typeid

3.6 final Keyword

The `final` keyword is a feature for C++11 and afterwards.

```
class A final{};
class B: public A {} //error!
```

It protect function being inherited. It can be also applied on virtual function to prevent further overriding.

3.7 Public, Private, and Protected inheritance

The public inheritance remains the property(public, protected, and private) of members in the base class. The **protected inheritance** shift public members in base class to protected, while the **private** shift everything in base class to private. In private and protected inheritance, slicing will not happen(is not allowed) as well.

4 Generic Programming

4.1 Function Template

Function template definition allows manipulations that are parameterized with different type variables. Based on the function template definition, the compiler will create a real "function"(**template instantiation**) that map the **formal parameter** to a real class type.

```
template <typename T>
inline const T& larger(const T& a, const T& b){return (a<b)?b:a;}
int main(){
    int result = larger(10,5);
    string result1 = larger("hello","world");
    int result2 = larger(1,1.1); // error! cannot decide type of T
    int result3 = <int>larger(10,5.5);
}
```

Note ⚠ Formal Argument Matching: explicit template instantiation or implicit template instantiation

It is important to note that the types accepted by the template should support the operations in the function. It is also important that the content to be applied to the function is proper. Template specialization is designed for certain types.

```
template<typename T>
const bool equal(const T& a, const T& b){return a==b;)
template<>
const bool equal(const float&a, const float&b){}
template<>
const bool equal(const double&a, const double&b){}
```

A template may take more than one type arguments, each using a different typename. They should be handled carefully as well!

The template mechanism works for classes as well. **⚠ Must specify the type first!**

```
template <typename T>
class List_Node{
    public:
        List_Node(const T& x) : data(x) { }
        T data;
        List_Node* next {nullptr};};
List_Node <int> a(5);
```

Template may have constants as **nontype**. They are usually considered as limitations.

4.2 Operator Overloading

We cannot compare user-defined Student type using built-in >. We need to define a (new) operator function as a member function of a new class, or as a global non-member function, where **operator** keyword specifies the operator.

```
Vector operator+(const Vector& a, const Vector& b)
 { return Vector(a.getx() + b.getx(), a.gety() + b.gety()); }
Vector a(1,2), b(2,0),c;
c = a + b;
```

- Overload operator+: global operator+ function is better: $a + b$ re-written as `operator+(a,b)`. While member operator function re-write a+b as `a.operator+(b)`(not always applicable for implicit conversion).

- Overload operator«:

Only existing operators(except 4 special operators) can be re-define, i.e. new symbols don't allowed. The number of operands, associativity, and precedence cannot be changed.