# C++总结2：

## 1：类和对象

- 0： 介绍
  - oop---面向对象的程序设计（Object Oriente Programming）
  - 面向对象三大特性：
    1 封装
    2 继承
    3 多态（重点）
  - 面向对象： 函数调用--->强调过程（自顶向下，逐步求精）
  - 面向过程： 以对象为中心，所有的操作由对象发出
  - 面向对象与面向过程的区别
- 1： 类
  - 0： 类的定义 —— 指的是一类事物，里面包括不同对象实体，它是一个抽象的概念；类用于抽象描述一类事物所共有的属性和行为，同城将属性和行为都用成员来表示
    - 成员包括：
      - 1： 属性 —— 指事物的静态体现----成员变量
      - 2： 行为 —— 指事物的动态体现----成员函数
  - 1： 类的封装
    - 封装的基本语法： class 类名 { };
    - 封装的目的： 封装类的成员，对类的成员起到隐蔽的作用
    - 封装的注意事项：
      - class封装的类，类中成员默认全部不公开。
      - struct封装的类，类中的成员全部是公开的。
      - 使用struct关键字和使用class关键字来封装类的区别
- 2： 类对象
  - 类对象创建
    - 成员访问方式： 对象 . 成员 指针->成员
    - 申请堆空间方式：
      - type* 指针变量名 = new type;
      - type* 指针变量名 = new type(初始化列表参数);
    - 堆内存释放： delete 指针变量名;
    - 扩展知识点：
      - new/delete和malloc/free的区别
        - 1： new/delete —— new/delete都是运算符，new申请堆内存，delete是释放堆内存，两者成对使用
        - 2： malloc/free —— nalloc/free都是函数，malloc申请堆内存（虚拟内存），free是释放堆内存，两者成对使用
        - new/delete相对于malloc/free的好处在于没有函数调用的栈开销，也不需要申请内存的大小，也不需要进行类型强转。new/delete在执行时，分别会自动调用构造析构函数，后者则不会，需要手动释放内存
      - delete与delete [] 的意义
        - 1： delete —— 只能释放一个对象
        - 2： delete [] —— 释放一个数组，一次将数组中的对象一个一个释放
- 3： 函数介绍
  - 3.1： 构造函数
    - 初步理解构造函数
      - 1 特点
        - 构造函数也是函数，但函数名跟类名相同
        - 构造函数无返回值
        - 构造函数可以重载
        - 构造函数创建对象时，会自动调用
      - 2 语法 —— 类名（参数列表）{ } --->参数通常是成员变量初始化参数值
      - 3 注意事项
        1. 当一个类在设计时，没有手动实现构造函数，则编译器会默认提供一个公开的一个无参构造函数---（默认构造）
        2. 若是设计者自定义了构造函数，不论该构造函数是有参还是无参，默认构造函数都不再存在
        3. 创建对象时，必须调用构造函数
    - 深入理解构造函数
      - 0 构造函数初始化列表
        - 定义： 使用一个初始化列表来初始类成员变量
        - 语法：

```
构造函数名(参数1,参数2,...,参数n):成员变量名1(参数1),成员变量2(参数2),...,
成员变量n(参数n)
{

}
```

        - 应用场景：
          1. 当一个类的成员变量由const修饰
          2. 在继承中，派生类构造函数中去初始化基类的成员变量
          3. 当类中的类成员变量是类类型时，使用构造函数初始化列表来初始化类类型中成员变量
      - 1 const 在C++扩展
        1. const修饰类成员变量--->必须使用构造函数初始化列表的方法进行初始化，且此后该成员变量就不能再修改，只能访问
        2. const修饰类成员函数
           - 1： 语法 —— 函数返回值类型 函数名（参数列表） const { //函数体 } ==> 类成员函数声明和实现时都要加const
           - 2： 注意事项
             1 在类中，类成员函数不会修改类成员变量时，都建议使用const来修饰
             2 const成员函数可以保证该次函数不会修改类成员变量的值，并且不会调用其他非const成员函数
             3 const成员函数可以构成重载
             4 当const成员函数构成重载时，则非const对象调用非const成员函数，const对象调用const成员函数
        3. const类对象
           - 定义： 即由const修饰的类类型对象
           - 目的： 保证在对象层面不会修改类的数据成员
  - 3.2： 拷贝构造函数 —— 语法=>
    - 浅拷贝：

```
类名（const 类名& other）
{
    成员变量 = other.成员变量;
}
```

      拷贝函数体可以修改
    - 注意事项：
      1. 当类成员变量中包含指针时，使用浅拷贝会使得两个对象的成员变量指针指向同一个地址
      2. 浅拷贝在进行堆空间释放时，会导致多次释放
    - 深拷贝：

```
//深拷贝
Test(const Test& other)
{
    _x = other._x;
    //深拷贝
    p = new int;
    *p = *(other.p);
}
```

    - 特性：
      1. 深拷贝使得创建对象时，让该对象的指针指向新的堆空间
      2. 在释放堆空间时，每个对象只需要释放自己申请的堆空间即可
    - 总结： 如果一个类的成员变量没有指针，不需要申请堆空间时，则可以直接使用默认的拷贝构造函数，若是有指针变量，则要手动进行深拷贝实现
  - 3.3： 析构函数
    - 0. 析构函数的语法

```
~Test()
{

}
```

    - 1. 析构函数的作用 —— 是为了更好的在对象被销毁时，做好清理和释放的工作
    - 2. 析构函数的特点
      0. 析构函数名与类名一致
      1. 析构函数不能构成重载，一个类只能有一个析构函数
      2. 析构函数无参
      3. 析构函数在对象被销毁时，自动调用
      4. 每一个对象被销毁时，就会自动调用一次析构函数
      5. 如果类设计者没有手动实现析构函数，编译器会提供一个默认的析构函数，析构体为空，一经手动实现，则系统默认的析构函数就不再存在
  - 3.4： 析构函数与深拷贝 —— 当类中有深拷贝时，在析构函数中要对指针进行释放

```
//析构函数
~Test()
{
    cout << " ~Test()" << endl;
    delete p;
}
```

- 4： 类成员函数和变量的实际存储方式
  - 存储方式：
    - 解析： 在创建一个类对象时，系统会为每一个对象分配存储空间。当一个类包括数据成员和成员函数，则要分别为数据和函数的代码分配存储空间
    - 引入：this指针
      - 1： 定义 —— 编译器为了有限内存存储，引入this指针，编译器将一个对象分为两个部分：1. 特有的（个性） 2. 公有的（共性），在使用公有的部分时，将具体的某一个对象也指定，如此就能知道那一个对象在调用公有部分
      - 2： 注意事项 —— 对于this指针，对象在调用成员函数（非static）时，并将对象的地址最隐含参数传入给函数，从而保证访问的成员是属于调用者
- 5： static修饰类成员变量
  - 1 作用
    - 实现同类对象之间的上诉局共享
    - 管理静态成员变量
    - 单实例模型-----可以使用静态成员函数来实现
  - 2 语法

```
class Test
{
  private:
  int a;
  int b;
  static int c;  //类内中声明一个static类成员变量
};
```

  - 3 注意事项
    1. 对于static类程员变量，如果 设置为public，破会了类的封装性，所以为了更好的管理static类成员变量，所以在C++中使用static类成员函数。
    2. 静态类成员变量是属于整个类，所以，在类的层面上看，是属于类本身的，但也可以通过对象和类名来调用
    3. 注意事项
       - 类外定义时，将static去掉
       - 对于静态成员函数内，this指针是失效的
       - 在静态成员函数中，不能访问非静态成员变量
       - 在静态成员函数中，不能访问非静态成员函数，厨房静态成员函数中的形参包含该类的对象
- 6： 继承
  - 1. 继承的语法
    - 0：定义 —— 继承的机制可自动地为一个类提供来自另一个类的操作和数据结构，这使得程序员只需要在新类中定义已有的类中没有的成分来建立一个新类
    - 1：继承解释 —— A类继承B类，则B类为“基类”，A为“子类”。A类继承B类之后，A类就有了B类的部分成员，具体得到的成员由两方面决定
      - 1 继承方式
        - * 公有继承（public）
          - **：该继承方式会把基类的公有成员（变量和函数）继承到子类公有成员，保护成员变为子类的保护成员，但是私有成员子类也一样不可访问
          - **：子类可以直接访问基类公共成员
          - ** 注意事项：
            - --基类的私有成员，子类不可以访问
            - --基类的保护成员，子类可以继承为自己的保护成员，在派生类可以访问，在外部不可以访问
            - --基类的公有成员，子类可以继承为自己的公有成员，在派生类中可以访问，在外部也可以访问
        - * 保护继承（protected）
          - **：保护继承方式会把基类的公有成员或者保护成员（变量和函数）变为子类的保护成员，但是私有成员子类也一样不可访问
          - **：子类无法直接访问基类的公有成员
          - **注意事项：
            - --基类共有成员，子类中继承为自己的保护成员，在派生类可以访问，在外部不可以访问
            - --基类保护成员，子类中继承为自己的保护成员，在派生类可以访问，在外部不可以访问
            - --基类私有成员，子类一样不可以访问基类的私有成员
        - * 私有继承（private）
          - **该继承方式指在继承时，把protected变成private
          - **注意事项：
            - --基类公有成员，子类中继承为自己的私有成员，在派生类可以访问，在外部不可以访问
            - --基类保护成员，子类中继承为自己的私有成员，在派生类可以访问，在外部不可以
            - --基类私有成员，子类一样不可以访问基类的私有成员
      - 2 基类成员的访问权限
    - 三种继承方式比较
      - 根据上述概况，可以得知：私有继承和保护继承作用完全相同。仔细分析其实还是有区别的，区别在于如果派生类再去派生其他类时，对于刚才的私有继承来说，再派生的类将得不到任何成员；而对于刚刚的保护继承，仍然能够得到基类的公有和保护成员。

| | A类(基类) | B类(A的派生类) | C类(B的派生类) |
|---|---|---|---|
| 公有继承 | 公有成员 | 公有成员 | 公有成员 |
| | 私有成员 | (无) | (无) |
| | 保护成员 | 保护成员 | 保护成员 |
| 私有继承 | 公有成员 | 私有成员 | (无) |
| | 私有成员 | (无) | (无) |
| | 保护成员 | 私有成员 | (无) |
| 保护继承 | 公有成员 | 保护成员 | 保护成员 |
| | 私有成员 | (无) | (无) |
| | 保护成员 | 保护成员 | 保护成员 |

      - 派生类的构造 - 派生类时可以访问基类保护的数据成员，但是还有一些私有数据成员，派生类时无法访问的，并且为提醒类的独立性，还是希望通过调用基类的成员函数去初始化这些变量，所以派生类是通过调用接力的构造函数，实现对成员变量的初始化。
  - 2. 继承中的构造与析构函数的调用
    - 例： 类名-> class GraduateStudent: public Student
    - （公有继承）构造-> GraduateStudent(string s , int g, int a):Student(s,g,a){ }
- 7： 多态
  - 1 定义 —— 函数的多种不同的实现方式即为多态
  - 2 必要性 —— 在继承中，有时候基类的一些哈哈你说在派生类中也是有用的，但是功能不够全，或者两者的功能实现方式就是不一样，此时就希望重载那个基类的函数，但是为了不调用该函数时出现不知道调用的是基类还是子类的情况出现，则提出了多态。如果语言不知多态，则不能称为面向对象的。
  - 3 多态的实现 —— 多态是依赖于虚函数来实现的，之所以虚函数可以分清楚当前调用的函数是基类还是派生类的，主要在于基类和派生类分别有着自己的虚函数表，再调用虚函数时，它们是通过去虚函数表去对应的函数的

```
#include<iostream>
using namespace std;


class A
{
public :
    void print()
    {
        cout << "A" <<< endl;
    }
    virtual void print1()
    {
        cout << "Virtual A" << endl;
    }
};

class B:public A{
public:
    void print() { cout << "B" << endl; }
    virtual void print1() { cout << "Virtual B" 
};

void fn(A & a)
{
    a.print();
    a.print1();
}
int main()
{

    A a;
    B b;
    fn(a);
    fn(b);
    return 0;
}
```

  - 4 注意事项 —— 若是基类和派生类有相同函数名的函数，但是参数不同，就会被看成两个不同函数。如果返回的类型不同，则也起不到多态的作用。还有一种情况，如果基类中虚函数返回一个基类指针或引用，派生类中返回一个派生类的指针或引用，则C++将其视为同名虚函数而进行迟后联编。

虚函数表的本质就是一种迟后联编的过程，正常编译都是先期联编，但是当代码遇到了virtual时，就会把它当做迟后联编，但是为了迟后编译，就生成了局部变量-虚函数表，这就会增大一些空间上的消耗（前提是两个函数的返回类型，参数类型，参数个数都相同，不然就不能起到多态作用）

- 使用虚函数的一些限制
  - 1：只有类成员函数才能声明为虚函数，因为虚函数只适用于有继承关系的类对象中
  - 2：静态成员函数不能说明为虚函数，因为静态成员函数不受限于某个对象，整个内存中只有一个，所以不会出现混淆情况
  - 3：内联函数不可以被继承，因为内联函数是不能从运行中动态的确认其位置
  - 4：构造函数不可以被继承
  - 5：析构函数可以被继承，而且通常声明为虚函数
- vtable:虚表

- 8：纯虚函数
  - **定义
    - 虚函数是在基类中被声明为virtual，并在派生类中重新定义的成员函数，可实现成员函数的动态重载
    - 纯虚函数的声明有着特殊的语法格式：virtual返回值类型成员函数名（参数表）=0；
  - **必要性 —— 在很多情况下，基类本身生成对象时不合情理的，为了解决该问题，映入了纯虚函数的概念（virtual ReturnType Function()=0;），则编译器要求在派生类中必须予以重载以实现多态性。
  - **抽象类
    - 定义： 行业纯虚函数的类即为抽象类。由于抽象类包含了没有定义的纯虚函数，所以不能定义抽象类的对象；在C++中，抽象类只能用于被继承而不能直接创建对象的类
    - 特点： 抽象类不能被实体化，但是可以声明指针或引用