C++核心编程

——C++面向对象编程

1 内存分区模型

C++程序执行时，将内存大方向划分四个区域

- 代码区：存放函数体的二进制代码，由操作系统进行管理的
- 全局区：存放全局变量和静态变量以及常量
- 栈区：由编译器自动分配释放，存放函数的参数值、局部变量等
- 堆区：由程序员分配和释放，若程序员不释放，程序结束时由操作系统回收

内存四区的意义：

不同区域存放的数据，赋予不同的生命周期，给我们更大的灵活编程

1.1 程序运行前

在程序编译后，生成了exe可执行程序，未执行该程序前分为两个区域：

代码区：

存放CPU执行的机器指令

特点：**共享、只读**

全局区：

全局变量和静态变量存放在此

全局区还包含了常量区，字符串常量和其他常量也存放在此

该区域的数据再程序结束后由操作系统释放

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

//全局变量
int g_a = 10;
int g_b = 10;
const int c_g_a = 10;
const int c_g_b = 10;
int main() {
    //全局区

    //普通局部变量
    int a = 10;
    int b = 10;
    cout << "局部变量a的地址：" << &a << endl;
    cout << "局部变量b的地址：" << &b << endl;

    cout << "全局变量g_a的地址：" << &g_a << endl;
    cout << "全局变量g_b的地址：" << &g_b << endl;

    //静态变量
    static int s_a = 10;
```

```
    static int s_b = 10;
    cout << "静态变量s_a的地址：" << &s_a << endl;
    cout << "静态变量s_b的地址：" << &s_b << endl;

    //常量
    //字符串常量
    cout << "字符串常量的地址：" << &"HelloWorld" << endl;
    //const修饰的变量
    //const修饰的全局变量
    cout << "全局常量c_g_a的地址：" << &c_g_a << endl;
    cout << "全局常量c_g_b的地址：" << &c_g_b << endl;
    //const 修饰的局部变量
    const int c_l_a = 10;
    const int c_l_b = 10;
    cout << "局部常量c_l_a的地址：" << &c_l_a << endl;
    cout << "局部常量c_l_b的地址：" << &c_l_b << endl;
    system("pause");
    return 0;
}
```

总结：

- C++在程序运行前分为全局区和代码区
- 代码区的特点是共享和只读
- 全局区中存放全局变量、静态变量、常量
- 常量区中存放const修饰的全局变量和字符串常量

1.2 程序运行后

栈区：

由编译器自动分配释放，存放函数的参数值、局部变量等

注意事项：不要返回局部变量的地址，栈区开辟的数据由编译器自动释放

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

//栈区的注意事项--不要返回局部变量的地址
//栈区的数据由编译器开辟释放

int* func() {
    int a = 10;//局部变量
    return &a;
}
int main() {
    cout << *(func()) << endl;
    int* p = func();//VS2022  分号结束  内存被释放
    cout << " *p = " << *p << endl;
    system("pause");
    return 0;
}
```

堆区：

由程序员分配释放，若程序员不释放，程序结束时由操作系统回收

在C++中主要利用new在堆区开辟内存

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

//在堆区开辟数据
int* func() {
    //利用new关键字，可以将数据开辟到堆区
    //指针本质也是局部变量，放在栈上，指针保存的内存是放在堆区的
    int* p = new int(10);
    return p;
}
int main() {
    int* p = func();
    cout << "*p = " << *p << endl;
    cout << "*p = " << *p << endl;
    system("pause");
    return 0;
}
```

1.3 new运算符

C++中利用new操作符在堆区开辟数据

堆区开辟的数据，由程序员手动开辟释放，释放利用操作符delete

语法：**new 数据类型**

利用new传概念的数据，会返回该数据对应的类型的指针

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

//new与delete
int* func() {
    //在堆区创建一个整形数据
    int* p = new int(10);
    return p;
}
void test01() {
    int* p = func();
    cout << *p << endl;
    //释放堆区内存
    delete p;
    //cout << *p << endl;//内存已被释放，再次访问属于非法操作
}
void test02() {
    //堆区开辟数组
    //创建10个整形数据的数组
    int* arr = new int[10];//返回首地址
    for (int i = 0; i < 10; i++) {
        arr[i] = i + 100;
    }
    for (int i = 0; i < 10; i++) {
        cout << arr[i] << "\t" << endl;
    }
    //释放堆区数组
    delete[] arr;
}
```

```cpp
int main() {
    test01();
    test02();
    system("pause");
    return 0;
}
```

2 引用

2.1 引用的基本使用

作用：给变量起别名

语法：数据类型 &别名 = 原名

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

int main() {
    //引用基本语法：数据类型 &别名 = 原名
    int a = 10;
    //创建引用
    int& b = a;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    b = 100;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    system("pause");
    return 0;
}
```

2.2 引用的注意事项

- 引用必须初始化
- 引用在初始化后，不可以改变

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

int main() {
    int a = 10;
    int& b = a;
    //引用的注意事项
    //1、引用必须初始化
    //int& c;    错误
    //2、引用在初始化后，不可以改变
    int c = 20;
    //int& b = c;    错误，重定义，多次初始化
    system("pause");
    return 0;
}
```

2.3引用做函数参数

作用：函数传参时，可以利用引用的技术让形参修饰实参

优点：可以简化指针修饰实参

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

//交换函数
//1、值传递
void Swap01(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
//2、地址传递
void Swap02(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
//3、引用传递
void Swap03(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
int main() {
    int a = 10;
    int b = 20;
    Swap01(a, b);//值传递，形参不会修饰实参
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    Swap02(&a, &b);//地址传递，形参会修饰实参
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    Swap03(a, b);//引用传递，形参会修饰实参
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    system("pause");
    return 0;
}
```

2.4 引用做函数返回值

作用：引用是可以作为函数的返回值存在的

注意：**不要返回局部变量的引用**

用法：函数调用作为左值

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

//引用做函数返回值
//1、不要返回局部变量的引用
```

```cpp
int& test01() {
    int a = 10;
    return a;
}
int& test02() {
    static int a = 10;//静态变量，存放在全局区
    return a;
}
int main() {
    //int& ref = test01();
    //cout << "ref = " << ref << endl;//结果错误，因为a的内存已经释放
    int& ref2 = test02();
    cout << "ref2 = " << ref2 << endl;
    //2、函数的调用可以作为左值
    test02() = 1000;//test02()返回a的引用，ref2是a的别名
    cout << "ref2 = " << ref2 << endl;

    system("pause");
    return 0;
    }
```

2.5 引用的本质

本质：**引用的本质在c++内部实现是一个指针常量**

```cpp
//发现是引用，转换成 int* const ref = &a;
void func(int& ref){
  ref = 100;//ref是引用，转换成*ref = 100
}
int main(){
  int a = 10;
  //自动转换成int* const ref = &a;指针常量是指针指向不可改
  int& ref = a;
  ref = 20;//内部发现ref是引用，自动转换成*ref = 20;
  return 0;
}
```

2.6 常量引用

作用：常量引用主要用来修饰形参，防止误操作

在函数形参列表中，可以加const修饰形参，防止形参改变实参

```cpp
#include <iostream>
using namespace std;
//打印数据函数
void ShowValue(const int& val) {
    //val = 1000;错误
    cout << "val = " << val << endl;
}
int main() {
    //常量引用
    //使用场景，用来修饰形参，防止误操作
    int a0 = 10;
    //加上const之后，编译器将代码修改，int temp = 10;const int &ref = temp;
    const int& ref = 10;//引用必须引一块合法的内存空间
```

```
    //ref = 20;加入const修饰后变成只读
    int a = 100;
    ShowValue(a);
    cout << "a = " << a << endl;

    system("pause");
    return 0;
}
```

3 函数提高

3.1 函数默认参数

在C++中，函数的形参列表中的形参是可以有默认值的

语法：**返回值类型 函数名(参数=默认值)**

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

//函数的默认参数
// 语法：返回值类型 函数名（形参 = 默认值）
//如果传入数据，掩盖原有的默认值

int func(int a,int b = 20,int c = 30) {
    return a + b + c;
}
//注意事项：
//1、如果某个位置已经有了默认参数，那么从这个位置往后，从左到右都必须有默认值
//int func2(int a, int b = 10, int c, int d);//默认实参不在形参列表结尾

//2、如果函数的声明有默认参数，函数实现不能有默认参数
//声明和实现只能有一个有默认参数
int func2(int a=10, int b=20);
//int func2(int a=10, int b=20) {重定义默认参数
//  return a + b;
//}
int func2(int a, int b) {
    return a + b;
}
int main() {
    cout << func(10) << endl;
    cout << func2() << endl;
    system("pause");
    return 0;
}
```

3.2 函数占位参数

C++的形参列表里可以有占位参数，用来做占位，调用函数时必须填补该位置

语法：返回值类型 函数名（数据类型）{}

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

```

```cpp
//占位参数
//返回值类型 函数名(数据类型){}
//占位参数可以有默认参数
void func(int a,int = 10) {
    cout << "func()调用" << endl;
}
int main()
{
    func(10,10);
    func(10);
    system("pause");
    return 0;
}
```

3.3 函数重载

3.3.1 函数重载概述

作用：函数名可以相同，提高复用性

函数重载满足条件：

- 同一个作用域下
- 函数名称相同
- 函数参数**类型不同**或者**个数不同**或者**顺序不同**

注意：函数的返回值不可以作为函数重载的条件

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

//函数重载条件
/*
    1、同一作用域
    2、函数名称相同
    3、函数参数类型不同或者个数不同或者顺序不同
*/
void func() {
    cout << "func()的调用" << endl;
}
void func(int a) {
    cout << "func(int)的调用" << endl;
}
void func(int a, double b) {
    cout << "func(int,double)的调用" << endl;
}
void func(double b,int a) {
    cout << "func(double,int)的调用" << endl;
}
//注意事项：函数的返回值不可以作为函数重载的条件
//int func(double b, int a) {
//  cout << "func(double,int)的调用" << endl;
//  return a;
//}
int main()
{
    func();
```

```
    func(10);
    func(10, 10.0);
    func(10.0, 10);
    system("pause");
    return 0;
}
```

3.3.2 函数重载注意事项

- 引用作为重载条件
- 函数重载碰到函数默认参数

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

//函数重载的注意事项
//1、引用作为重载的条件
void func(int &a) {
    cout << "func(int &a)调用" << endl;
}
void func(const int& a) {
    cout << "func(const int &a)调用" << endl;
}
//2、函数重载碰到默认参数
void func2(int a,int b = 10) {
    cout << "func2(int a,int b = 10)调用" << endl;
}
void func2(int a) {
    cout << "func2(int a)调用" << endl;
}
int main()
{
    int a = 10;
    func(a);//a可读可写，调用无const的版本
    func(10);//10只读,int &a = 10,不合法;const int &a = 10,合法;
//  func2(10);当函数重载碰到默认参数，出现二义性，尽量避免
    func2(10, 20);

    system("pause");
    return 0;
}
```

4 类和对象

C++面向对象的三大特性：**封装、继承、多态**

C++认为**万事万物皆为对象**，对象上有其属性和行为

具有相同性质的**对象**，抽象为**类**

4.1 封装

4.1.1封装的意义

封装是C++面向对象的三大特性之一

封装的意义：

- 将属性和行为作为一个整体
- 降属性和行为加以权限控制

封装意义一：

在设计类的时候，属性和行为写在一起，表现事物

语法：class 类名{访问权限：属性 / 行为};

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

const double PI = 3.14;
//设计一个圆类，求圆的周长
class Circle {
//访问权限
public:
    //属性
    int m_r;//半径
    //行为
    double calculateZC() {  //获取圆的周长
        return 2 * PI * m_r;
    }
};
int main()
{
    Circle c1;//创建圆对象
    c1.m_r = 10;
    cout << "圆的周长为：" << c1.calculateZC() << endl;
    system("pause");
    return 0;
}
```

- 设计一个学生类，属性有姓名和学号
- 可以给姓名和学号复制，可以显示学生的姓名和学号

```
#include <iostream>
#include <string>
using namespace std;
//学生类
class Student {
public:
    //类中的属性和行为同一称为成员
    //属性：成员属性/成员变量
    //行为：成员函数/成员方法
    void SetName(string name) {
        m_name = name;
    }
    void SetId(int id) {
        m_id = id;
    }
    void ShowStudent() {
```

```
        cout << "姓名: " << m_name << "\t学号: " << m_id << endl;
    }
    string m_name;
    int m_id;

};
int main()
{
    Student stu1;
    string name;
    int id;
    cout << "请输入姓名: ";
    cin >> name;
    cout << endl;
    cout << "请输入学号: ";
    cin >> id;
    cout << endl;
    stu1.SetName(name);
    stu1.SetId(id);
    stu1.ShowStudent();
    system("pause");
    return 0;
}
```

封装意义二:

类在设计时，可以把属性和行为放在不同的权限下，加以控制

访问权限有三种:

- public 公共权限
- protected 保护权限
- private 私有权限

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

//访问权限
//公共权限 public        成员类内类外皆可访问
//保护权限 protected     成员类内可以访问，类外不可以访问,子类可以访问父类中的保护权限
//私有权限 private       成员类内可以访问，类外不可以访问,子类不可以访问父类中的保护权限
class Person {
public:
    string m_Name;//姓名
protected:
    string m_Car;//汽车
private:
    int m_PassWord;//银行卡密码

public:
    //类内访问
    void func() {
        m_Name = "Tom";
        m_Car = "BMW";
        m_PassWord = 123;
    }
};
```

```cpp
int main()
{
    Person p1;//实例化具体对象
    p1.m_Name = "Jerry";
    p1.func();
    //类外不可访问保护权限和私有权限内容
    //p1.m_Car = "拖拉机";
    //p1.m_PassWord = 456;
    system("pause");
    return 0;
}
```

4.1.2 class和struct的区别

在C++中struct和class唯一的**区别**就在于**默认的访问权限不同**

区别：

- struct默认权限为公共
- class默认权限为私有

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

class C1 {
    int m_A;//默认权限是私有
};
struct C2{
    int m_A;//默认权限是公共
};
int main()
{
    C1 c1;
    //c1.m_A = 10;私有权限类外不可访问
    C2 c2;
    c2.m_A = 10;
    system("pause");
    return 0;
}
```

4.1.3 成员属性设置为私有

优点一：将所有成员属性设置为私有，可以自己控制读写权限、

优点二：对于写权限，我们可以检测数据的有效性

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

//成员属性设置为私有
class Person {
public:
     //提供接口
    void SetName(string name) {
        m_Name = name;
```

```
    }
    string GetName() {
        return m_Name;
    }
    void SetAge(int age) {
        //判断数据的有效性
        if (!(age > 0 && age < 150)) {
            cout << "非法输入！" << endl;
            return;
        }
        m_Age = age;
    }
    int GetAge() {
        return m_Age;
    }
    void SetLover(string lover) {
        m_Lover = lover;
    }
private:
    string m_Name;//姓名---可读可写
    int m_Age;//年龄---只读
    string m_Lover;//情人---只写
};
int main()
{
    Person p;
    p.SetName("张三");
    cout << "姓名：" << p.GetName() << endl;
    p.SetAge(75);
    cout << "年龄：" << p.GetAge() << endl;
    p.SetLover("李四");
    system("pause");
    return 0;
}
```

- 封装案例一：设计立方体类

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

//设计立方体类
//求出立方体的面积和体积
//分别用全局函数和成员函数来判断两个立方体是否相等
class Cube {
public:
    //设置长
    void SetL(double l) {
        m_L = l;
    }
    //获取长
    double GetL() {
        return m_L;
    }
    //设置宽
    void SetW(double w) {
        m_W = w;
```

```
    }
    //获取宽
    double GetW() {
        return m_W;
    }
    //设置高
    void SetH(double h) {
        m_H = h;
    }
    //获取高
    double GetH() {
        return m_H;
    }
    //获取立方体面积
    double CalculateS() {
        return 2 * (m_L * m_W + m_L * m_H + m_W * m_H);
    }
    //获取立方体面积
    double CalculateV() {
        return m_L * m_W * m_H;
    }
    //成员函数判断
    bool isSameByClass(Cube& c) {
        if (m_L == c.GetL() && m_W == c.GetW() && m_H == c.GetH()) {
            return true;
        }
        return false;
    }
private:
    double m_L; //长
    double m_W; //宽
    double m_H; //高
};

//全局函数判断
bool isSame(Cube& c1, Cube& c2) {
    if (c1.GetL() == c2.GetL() && c1.GetW() == c2.GetW() && c1.GetH() ==
c2.GetH()) {
        return true;
    }
    return false;
}
int main()
{
    Cube c1;//实例化立方体对象
    c1.SetL(10.0);
    c1.SetW(10.0);
    c1.SetH(10.0);
    cout << "c1的面积为：" << c1.CalculateS() << endl;
    cout << "c1的体积为：" << c1.CalculateV() << endl;
    Cube c2;//实例化立方体对象
    c2.SetL(10.0);
    c2.SetW(10.2);
    c2.SetH(10.0);
    //bool ret = isSame(c1, c2);
    bool ret = c1.isSameByClass(c2);
    if (ret) {
        cout << "c1 == c2" << endl;
```

```
    }
    else {
        cout << "c1 != c2" << endl;
    }
    system("pause");
    return 0;
}
```

- 封装案例二：点和圆的关系

```
//main.cpp
#include <iostream>
#include <string>
using namespace std;
#include"point.h"
#include"circle.h"
//判断点和圆的关系
//判断
void isInCircle(Circle& c, Point& p) {
    //计算两点间距离的平方
    int distence = (c.GetCenter().GetX() - p.GetX()) * (c.GetCenter().GetX() - p.GetX()) +
        (c.GetCenter().GetY() - p.GetY()) * (c.GetCenter().GetY() - p.GetY());
    //计算半径的平方
    int rDistence = c.GetR() * c.GetR();
    //判断关系
    if (distence == rDistence) {
        cout << "点在圆上" << endl;
    }
    else if (distence > rDistence) {
        cout << "点在圆外" << endl;
    }
    else {
        cout << "点在圆内" << endl;
    }
}
int main()
{
    //实例化圆
    Circle c;
    c.SetR(10);
    Point center;
    center.SetX(10);
    center.SetY(0);
    c.SetCenter(center);
    //实例化点
    Point p;
    p.SetX(10);
    p.SetY(5);
    //判断关系
    isInCircle(c, p);
    system("pause");
    return 0;
}
```

```cpp
//point.h
#pragma once
#include <iostream>
using namespace std;
//点类
class Point {
private:
    int m_X;
    int m_Y;
public:
    //设置获取X
    void SetX(int x);
    int GetX();
    //设置获取Y
    void SetY(int y);
    int GetY();
};
```

```cpp
//point.cpp
#include "point.h"
//设置获取X
void Point::SetX(int x) {
    m_X = x;
}
int Point::GetX() {
    return m_X;
}
//设置获取Y
void Point::SetY(int y) {
    m_Y = y;
}
int Point::GetY() {
    return m_Y;
}
```

```cpp
//circle.h
#pragma once
#include <iostream>
using namespace std;
#include "point.h"
//圆类
class Circle {
public:
    //设置获取半径
    void SetR(int r);
    int GetR();
    //设置获取圆心
    void SetCenter(Point center);
    Point GetCenter();
private:
    int m_R;//半径
    Point m_Center;
};
```

```cpp
//circle.cpp
```

```cpp
#include "circle.h"
//设置获取半径
void Circle::SetR(int r) {
    m_R = r;
}
int Circle::GetR() {
    return m_R;
}
//设置获取圆心
void Circle::SetCenter(Point center) {
    m_Center = center;
}
Point Circle::GetCenter() {
    return m_Center;
}
```

总结：

- 一个类可以作为另一个类的成员
- 按逻辑拆分代码,头文件中写声明，cpp文件中实现

4.2 对象的初始化和清理

4.2.1 构造函数和析构函数

对象的**初始化和清理**是两个非常重要的安全问题

C++中**构造函数**和**析构函数**将会被编译器自动调用，完成对象初始化和清理工作，对象的初始化和清理工作是编译器强制我们做的事。因此如果我们不提供构造和析构，编译器会提供。编译器提供的构造函数和析构函数是空实现。

- 构造函数：主要作用在于创建对象时为对象的成员属性赋值，构造函数由编译器自动调用，无需手动调用。
- 析构函数：主要作用在于对象**销毁前**系统自动调用，执行一些清理工作

构造函数语法：类名（）{}

- 构造函数：没有返回值也不写void
- 函数名与类名相同
- 构造函数可以有参数，因此可以发生重载
- 程序在调用对象时候会自动调用构造，无需手动调用而且只会调用一次

析构函数语法：~类名（）{}

- 析构函数：没有返回值也不写void
- 函数名与类名相同，在名称前加上符号~
- 构造函数不可以有参数，因此补可以发生重载
- 程序在对象销毁前会自动调用析构，无需手动调用而且只会调用一次

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

//对象的初始化和清理
class Person {
public:
    //1、构造函数
    Person() {
        cout << "Person()构造函数调用" << endl;
```

```
    }

    //2、析构函数
    ~Person() {
        cout << "~Person()析构函数调用" << endl;
    }
};
void test01() {
    Person p;
}
int main()
{
    test01();
    //Person p;
    system("pause");
    return 0;
}
```

4.2.2 构造函数的分类及调用

两种分类方式：

- 按参数分为：有参构造和无参构造
- 按类型分为：普通构造和拷贝构造

三种调用方式：

- 括号法
- 显示法
- 隐式转换法

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

//构造函数的分类及调用
//分类
//按参数分类：无参构造（默认构造）和有参构造
//按类型分类：普通构造 拷贝构造
class Person {
public:
    //构造函数
    Person() {
        m_Age = 18;
        cout << "Person的无参构造函数调用" << endl;
    }
    Person(int a) {
        m_Age = a;
        cout << "Person的有参构造函数调用" << endl;
    }
    //拷贝构造函数
    Person(const Person& p) {
        //将传入的类的属性拷贝
        m_Age = p.m_Age;
        cout << "Person的拷贝构造函数调用" << endl;

    }
    int m_Age;
    //析构函数
```

```
    ~Person() {
        cout << "Person的析构函数调用" << endl;
    }
};
//调用
void test01() {
    //1、括号法
    Person p1;          //默认构造函数的调用
    Person p2(18);      //有参构造函数的调用
    Person p3(p2);      //拷贝构造函数的调用
    //注意事项：
    //1、调用默认构造函数时不要加()
    //Person p1();编译器认为这是函数声明，不会认为在创建对象

    //2、显示法
    Person p4 = Person(18);//有参构造函数的调用
    Person p5 = Person(p2);//拷贝构造函数的调用
    //Person(18)---匿名对象，特点：当前行执行结束后，系统会立即回收掉匿名对象
    //注意事项：
    //2、不要利用拷贝构造函数初始化匿名对象
    //Person(p4);编译器认为Person(p4) === Person p4,对象声明，重定义

    //3、隐式转换法
    Person p6 = 10;//有参构造函数调用    Person p4 = Person(18);
    Person p7 = p6;//拷贝构造函数调用
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

4.2.3 拷贝构造函数调用时机

C++调用拷贝构造函数调用时机：

- 使用一个已经创建完毕的对象来初始化一个新对象
- 值传递的方式给函数参数传值
- 以值方式返回局部对象

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

//拷贝构造函数的调用时机

class Person {
public:
    Person() {
        cout << "Person默认构造函数调用" << endl;
    }
    Person(int age) {
        cout << "Person有参构造函数调用" << endl;
        m_Age = age;
    }
    Person(const Person& p) {
        cout << "Person拷贝构造函数的调用" << endl;
```

```
        m_Age = p.m_Age;
    }
    ~Person() {
        cout << "Person析构函数调用" << endl;
    }
    int m_Age;
};
//1、使用一个已经创建完毕的对象来创建一个新对象
void test01() {
    Person p1(20);
    Person p2(p1);
    cout << "p2.m_Age = " << p2.m_Age << endl;
}
//2、值传递方式给函数参数传值
void doWork(Person p) {
    //值传递的本质：拷贝一份副本数据
}
void test02() {
    Person p;
    doWork(p);
}
//3、值方式返回局部对象
Person doWork2() {
    Person p1;
    cout << "&p1 = " << &p1 << endl;
    return p1;//返回一个拷贝的p1
}
void test03() {
    Person p = doWork2();
    cout << "&p = " << &p << endl;
}
int main()
{
    test01();
    test02();
    test03();
    system("pause");
    return 0;
}
```

4.2.4 构造函数调用规则

默认情况下，C++编译器至少给一个类添加3个函数

- 默认构造函数（无参，函数体为空）
- 默认析构函数（无参，函数体为空）
- 默认拷贝构造函数，对属性进行值拷贝

构造函数调用规则：

- 如果用户定义有参构造函数，C++不再提供默认无参构造，但是会提供默认拷贝构造
- 如果用户定义拷贝构造函数，C++不再提供其他构造函数

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

//构造函数调用规则

```

```
class Person {
public:
    //无参（默认）构造函数
    Person() {
        cout << "person无参构造函数调用" << endl;
    }
    //有参构造函数
    Person(int age) {
        cout << "Person有参构造函数调用" << endl;
        m_Age = age;
    }
    //拷贝构造函数
    Person(const Person& p) {
        cout << "Person拷贝构造函数调用" << endl;
        m_Age = p.m_Age;
    }

    ~Person() {
        cout << "Person析构函数调用" << endl;
    }
    int m_Age;
};
void test01() {
    Person p;
    p.m_Age = 18;

    Person p2(p);
    cout << "p2.m_Age = " << p2.m_Age << endl;
}
void test02() {
    //如果用户定义有参构造函数，C++不再提供默认无参构造，但是会提供默认拷贝构造
    //如果用户定义拷贝构造函数，C++不再提供其他构造函数
    Person p1(19);
    Person p2(p1);
}

int main()
{
    test01();
    test02();
    system("pause");
    return 0;
}
```

4.2.5 深拷贝和浅拷贝

- 浅拷贝：简单的复制拷贝操作
- 深拷贝：在对去重新申请空间，进行拷贝操作

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

//深拷贝和浅拷贝
class Person {
public:
    Person() {
        cout << "Person的默认构造函数调用" << endl;
```

```
    }
    Person(int age,int height) {
        m_Age = age;
        m_Height = new int(height);
        cout << "Person的有参构造函数调用" << endl;
    }
    Person(const Person& p) {
        m_Height = new int(*p.m_Height);
        m_Age = p.m_Age;
        cout << "Person拷贝构造函数的调用" << endl;
    }
    ~Person() {
        if (m_Height != NULL) {
            delete m_Height;
            m_Height = NULL;
        }
        cout << "Person的析构函数调用" << endl;
    }
    int m_Age;
    int* m_Height;
};
void test01() {
    Person p1(18,160);
    cout << "p1.m_Age = " << p1.m_Age << endl;
    cout << "*p1.m_Height = " << *p1.m_Height << endl;
    //编译器提供的拷贝构造函数做的是浅拷贝的操作
    //浅拷贝带来的问题是堆区内存重复释放------深拷贝解决
    Person p2(p1);
    cout << "p2.m_Age = " << p2.m_Age << endl;
    cout << "*p1.m_Height = " << *p2.m_Height << endl;

}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

4.2.6 初始化列表

作用：初始化属性

语法：构造函数():属性1(值1),属性2(值2),.....{};

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

//初始化列表
class Person {
public:

    //传统初始化操作
    /*Person(int a,int b,int c) {
        m_A = a;
        m_B = b;
        m_C = c;
```

```
    }*/
    //初始化列表
    Person(int a,int b,int c) :m_A(a), m_B(b), m_C(c) {

    }
    int m_A;
    int m_B;
    int m_C;
};
void test01() {
    Person p(10, 20, 30);
    cout << "m_A = " << p.m_A << endl;
    cout << "m_B = " << p.m_B << endl;
    cout << "m_C = " << p.m_C << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

4.2.7 类对象作为类成员

C++类中的成员可以使另一个类的对象，该成员为对象成员

例如：

```
class A;
class B{
  A a;//A为对象成员
}
```

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

//类对象作为类成员
class Phone {
public:
    Phone(string pName) {
        cout << "Phone有参构造函数调用" << endl;
        m_PName = pName;
    }
    ~Phone() {
        cout << "Phone析构函数调用" << endl;
    }
    string m_PName;
};
class Person{
public:
    //Phone m_Phone = pName;隐式转换法
    Person(string name, string pName) :m_Name(name), m_Phone(pName) {
        cout << "Person有参构造函数调用" << endl;
    }
    ~Person() {
        cout << "Person析构函数调用" << endl;
```

```
    }
    string m_Name;
    Phone m_Phone;
};
//当其他类对象作为本类成员，先构造其他类，在构造自身
//析构顺序与构造顺序相反
void test01() {
    Person p("Tom", "iPhone");
    cout << p.m_Name << " with " << p.m_Phone.m_PName << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

4.2.8 静态成员

静态成员就是在成员变量和成员函数前加上关键字static，称为静态成员

静态成员分为：

- 静态成员变量
  - 所有对象共享同一份数据
  - 在编译阶段分配内存
  - 类内声明，类外初始化

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

//静态成员变量
class Person {
public:
    static int m_A;//类内声明
    //静态成员拥有访问权限
private:
    static int m_B;
};
int Person::m_A = 100;//类外初始化
int Person::m_B = 200;
void test01() {
    Person p;
    cout << "p.m_A = " << p.m_A << endl;
    //cout << "p.m_B = " << p.m_B << endl;类外不可访问
    Person p2;
    p2.m_A = 200;
    cout << "p.m_A = " << p.m_A << endl;
}
void test02() {
    //静态成员不属于某个对象上
    //因此静态成员变量有两种访问方式

    //1、通过对象访问
    Person p;
    cout << "p.m_A = " << p.m_A << endl;
    //2、通过类名访问
```

```
    cout << "Person::m_A = " << Person::m_A << endl;
}
int main()
{
    //test01();
    test02();
    system("pause");
    return 0;
}
```

- 静态成员函数
  - 所有对象共享用一个函数
  - 静态成员函数只能访问静态成员变量

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

//静态成员函数
class Person {
public:
    static void func() {
        m_A = 100;//静态成员函数可以访问静态成员变量
        //m_B = 200;静态成员函数不可以访问静态成员变量，无法区分特定变量的成员
        cout << "static void func()调用" << endl;
    }
    static int m_A;
    int m_B;
    //静态成员函数拥有访问权限
private:
    static void func2() {
        cout << "static void func2()调用" << endl;
    }
};
int Person::m_A = 0;
void test01() {
    //访问静态成员函数
    //1、通过对象访问
    Person p;
    p.func();
    //2、通过类名访问
    Person::func();
    //Person::func2();类外不可访问
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

4.3 C++对象模型和this指针

4.3.1 成员变量和成员函数分开存储

C++中，类内的成员变量和成员函数是分开存储的，只有非静态成员变量才属于类的对象

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

//成员变量和成员函数分开存储
class Person {
public:
    int m_A;//非静态成员变量属于类的对象
    static int m_B;//静态成员变量不属于类的对象
    void func() {};//非静态成员函数不属于类的对象
    static void func2() {};//静态成员函数不属于类的对象
};
int Person::m_B = 100;
void test01() {
    Person p;
    //空对象占用内存空间为1
    //编译器是为了区分空对象占内存的位置，每个空对象有特定的内存空间
    cout << "sizeof(p) = " << sizeof(p) << endl;
}
void test02() {
    Person p;
    cout << "sizeof(p) = " << sizeof(p) << endl;
}
int main()
{
    test01();
    test02();
    system("pause");
    return 0;
}
```

4.3.2 this指针概念

this指针指向被调用的成员函数所属的对象

this指针是隐含每一个非静态成员函数内的一种指针，无需定义，直接使用

this指针的用途：

- 当形参和成员变量同名是，可用this指针来区分
- 当类的非静态成员函数中返回对象本身，可以使用return *this

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

//this指针
class Person {
public:
    Person(int age) {
        //this指针指向被调用的成员函数所属的对象
        this->m_Age = age;
    }
```

```
    Person &PersonAddAge(Person& p) {
        this->m_Age += p.m_Age;
        return *this;
    }
    int m_Age;
};
//1、解决名称冲突
void test01() {
    Person p1(18);
    cout << "p1.m_Age = " << p1.m_Age << endl;
}
//2、返回对象本身
void test02() {
    Person p1(10);
    Person p2(10);
    //链式编程思想
    p2.PersonAddAge(p1).PersonAddAge(p1).PersonAddAge(p1);
    cout << "p2.m_Age = " << p2.m_Age << endl;
}

int main()
{
    //test01();
    test02();
    system("pause");
    return 0;
}
```

4.3.3 空指针访问成员函数

C++中空指针也是可以调用成员函数的，但是要注意有没有用到this指针

如果用到this指针，需要加以判断代码的健壮性

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

//空指针调用成员函数
class Person {
public:
    void showClassName() {
        cout << "This is Person Class" << endl;
    }

    void showPersonAge() {
        //报错原因是因为传入指针为空
        if (this == NULL) { return; }
        cout << "m_Age = " << this->m_Age << endl;
    }
    int m_Age;
};

void test01() {
    Person* p = NULL;
    p->showClassName();
    //p->showPersonAge();
}
```

```
24
25  int main()
26  {
27      test01();
28      system("pause");
29      return 0;
30  }
```

4.3.4 const修饰成员函数

常函数:

- 成员函数后加const后的函数称为常函数
- 常函数内不可以修改成员属性
- 成员属性声明是加关键字mutable后，在常函数中依然可以修改

常对象:

- 声明对象前加const的对象称为常对象
- 常对象只能调用常函数

```
1   #include <iostream>
2   using namespace std;
3
4   //常函数 //常对象
5   class Person {
6   public:
7       //this指针的本质是指针常量，指向不可更改
8       //常函数修饰的this指针
9       void showPerson() const {
10          //常函数的const-->this指针变为：const Person *const this;
11          this->m_B = 100;
12      }
13      void func() {}
14      int m_A;
15      mutable int m_B;//mutable修饰，可在常函数中更改,常对象也可更改
16  };
17
18  void test01() {
19      Person p;
20      p.showPerson();
21  }
22  void test02() {
23      const Person p;//常对象
24      //p.m_A = 100;
25      p.m_B = 100;
26      //常对象只能调用常函数
27      p.showPerson();
28      //p.func();
29  }
30  int main()
31  {
32      test01();
33      test02();
34      system("pause");
35      return 0;
36  }
```

4.4 友元

友元的目的：让一个函数或者类访问另一个类中私有成员

友元的关键字**friend**

友元的三种实现：

- 全局函数做友元
- 类做友元
- 成员函数做友元

4.4.1 全局函数做友元

```
#include <iostream>
using namespace std;
#include <string>
//全局函数做友元
class Building {
    friend void GoodGay(Building* building);//友元全局函数
public:
    Building() {
        m_SittingRoom = "客厅";
        m_BedRoom = "卧室";
    }
    string m_SittingRoom;
private:
    string m_BedRoom;
};
void GoodGay(Building* building) {
    cout << "GoodGay()正在访问" << building->m_SittingRoom << endl;
    cout << "GoodGay()正在访问" << building->m_BedRoom << endl;
}
void test01() {
    Building building;
    GoodGay(&building);
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

4.4.2 类做友元

```
#include <iostream>
using namespace std;
#include <string>
//类做友元
class Building {
    friend class GoodGay;//友元类
public:
    Building();
    string m_SittingRoom;
private:
    string m_BedRoom;
```

```
};
class GoodGay {
public:
    GoodGay();
    void visit();
    Building* building;
};
//类外实现成员函数
Building::Building() {
    m_BedRoom = "卧室";
    m_SittingRoom = "客厅";
}
GoodGay::GoodGay() {
    building = new Building;
}
void GoodGay::visit() {
    cout << "GoodGay()正在访问" << building->m_SittingRoom << endl;
    cout << "GoodGay()正在访问" << building->m_BedRoom << endl;
}

void test01() {
    GoodGay gg;
    gg.visit();
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

4.4.3 成员函数做友元

```
#include <iostream>
using namespace std;
#include <string>
//成员函数做友元
class Building;
class GoodGay {
public:
    GoodGay();
    void visit();//友元函数
    void visit2();
    Building* building;
};
class Building {
    friend void GoodGay::visit();//友元函数
public:
    Building();
    string m_SittingRoom;
private:
    string m_BedRoom;
};
Building::Building() {
    m_BedRoom = "卧室";
    m_SittingRoom = "客厅";
}
```

```
GoodGay::GoodGay() {
    building = new Building;
}
void GoodGay::visit() {
    cout << "GoodGay::visit()正在访问" << building->m_SittingRoom << endl;
    cout << "GoodGay::visit()正在访问" << building->m_BedRoom << endl;
}
void GoodGay::visit2() {
    cout << "GoodGay::visit2()正在访问" << building->m_SittingRoom << endl;
}
void test01() {
    GoodGay gg;
    gg.visit();
    gg.visit2();
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

4.5 运算符重载

- 使更多的数据类型适用运算符

4.5.1 加号运算符重载

作用：实现两个自定义数据类型相加的运算

```
#include <iostream>
using namespace std;
#include <string>
//加号运算符重载
class Person {
public:
    //成员函数重载+
    /*Person operator+(Person&p) {
        Person temp;
        temp.m_A = this->m_A + p.m_A;
        temp.m_B = this->m_B + p.m_B;
        return temp;
    }*/
    int m_A;
    int m_B;
};
//全局函数重载+
Person operator+(Person& p1, Person& p2) {
    Person temp;
    temp.m_A = p1.m_A + p2.m_A;
    temp.m_B = p1.m_B + p1.m_B;
    return temp;
}
//函数重载的版本
Person operator+(Person& p1,int num) {
    Person temp;
    temp.m_A = p1.m_A + num;
```

```
    temp.m_B = p1.m_B + num;
    return temp;
}
void test01() {
    Person p1;
    p1.m_A = 10;
    p1.m_B = 20;
    Person p2;
    p2.m_A = 10;
    p2.m_B = 20;

    Person p3 = p1 + p2;
    //运算符重载也可以发生函数重载
    Person p4 = p1 + 10;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

总结:

- 对于内置的数据类型表达式的运算符是不可能改变的
- 不要滥用运算符重载

4.5.2 左移运算符重载

```
#include <iostream>
using namespace std;
#include <string>
//左移运算符重载
class Person {
    friend ostream& operator<<(ostream& cout,Person& p);
public:
    Person(int a,int b){
        m_A = a;
        m_B = b;
    }
private:
    int m_A;
    int m_B;
};
//全局函数重载<<,成员函数重载<<运算符，无法实现，因为cout在左侧
ostream& operator<<(ostream& cout,Person& p) {//本质 operator << (cout,p)--
>cout<<pcout << "p.m_A = " << p.m_A << endl;
    cout << "p.m_A = " << p.m_A << endl;
    cout << "p.m_B = " << p.m_B << endl;
    return cout;//实现链式编程
}
void test01() {
    Person p(10,10);
    cout << p << endl;//本质 operator<<(cout, p);
}
int main()
{
```

```
    test01();
    system("pause");
    return 0;
}
```

4.5.3 递增运算符重载

作用：通过重载递增运算符，实现自己的整形数据

```
#include <iostream>
using namespace std;
//递增运算符重载
class MyInteger {
    friend ostream& operator<<(ostream& cout, MyInteger myint);
public:
    MyInteger() {
        m_Num = 0;
    }
    //前置递增
    MyInteger& operator++() {
        //先递增
        m_Num++;
        //再将自身返回,返回引用，一直对一个数操作
        return *this;
    }
    //后置递增,int 代表占位参数，用于区分前置和后置递增
    MyInteger operator++(int) {
        //先记录当时结果
        MyInteger temp = *this;
        //再递增
        m_Num++;
        //再将记录的结果返回，返回值（不可返回局部对象的引用）
        return temp;
    }
private:
    int m_Num;
};
//重载<<
ostream& operator<<(ostream& cout, MyInteger myint) {
    cout << myint.m_Num;
    return cout;
}
void test01() {
    MyInteger myint;
    cout << ++(++myint) << endl;
    cout << myint << endl;
}
void test02() {
    MyInteger myint;
    cout << myint++ << endl;
    cout << myint << endl;
}
int main() {
    test01();
    test02();
    system("pause");
    return 0;
```

```
}
```

- 重载递减运算符

```
#include <iostream>
using namespace std;
//递减运算符重载
class MyInter {
    friend ostream& operator<<(ostream& cout, MyInter myint);
private:
    int m_Num;
public:
    MyInter() {
        m_Num = 0;
    }
    MyInter& operator--() {
        m_Num--;
        return *this;//返回自身
    }
    MyInter operator--(int) {
        MyInter temp = *this;//先保存当前的值
        m_Num--;
        return temp;
    }
};
ostream& operator<<(ostream&cout,MyInter myint) {
    cout << myint.m_Num;
    return cout;
}
void test01() {
    MyInter myint;
    cout << --(--myint) << endl;
    cout << (myint--)-- << endl;
}
int main() {
    test01();
    system("pause");
    return 0;
}
```

4.5.4 赋值运算符重载

C++编译器给一个类添加4个函数

- 默认构造函数（无参，函数体为空）
- 默认析构函数（无参，函数体为空）
- 默认拷贝构造函数，对属性进行值拷贝
- 复制运算符operator=,对属性进行拷贝（有属性指向堆区，做赋值操作时会出现深浅拷贝的问题）

```
#include <iostream>
using namespace std;
//赋值运算符重载
class Person {
public:
    Person(int age) {
        m_Age = new int(age);
```

```
    }
    Person& operator=(Person& p) {
        //m_Age = p.m_Age;编译器提供
        //先判断堆区是否干净
        if (m_Age != NULL) {
            delete m_Age;
            m_Age = NULL;
        }
        this->m_Age = new int(*p.m_Age);//深拷贝
        return *this;//返回对象本身，链式编程
    }
    ~Person() {
        if (m_Age != NULL) {
            delete m_Age;
            m_Age = NULL;
        }
    }
    int* m_Age;
};
void test01() {
    Person p1(18);
    Person p2(19);
    Person p3(30);
    p3 = p2 = p1;
    cout << "*p1.m_Age = " << *p1.m_Age << endl;
    cout << "*p2.m_Age = " << *p2.m_Age << endl;
    cout << "*p3.m_Age = " << *p3.m_Age << endl;
}
int main() {
    test01();
    system("pause");
    return 0;
}
```

4.5.5 关系运算符重载

作用：重载关系运算符，让两个自定义类型对象进行对比操作

```
#include <iostream>
using namespace std;
#include <string>
//关系运算符重载
class Person {
public:
    Person(string name,int age) {
        m_Age = age;
        m_Name = name;
    }
    bool operator==(Person& p) {
        if (p.m_Name == this->m_Name && p.m_Age == this->m_Age) {
            return true;
        }
        return false;
    }
    bool operator!=(Person& p) {
        if (p.m_Name == this->m_Name && p.m_Age == this->m_Age) {
            return false;
```

```
        }
        return true;
    }
    string m_Name;
    int m_Age;
};
void test01() {
    Person p1("Tom", 18);
    Person p2("Tom", 18);
    if (p1 != p2) {
        cout << "p1 != p2" << endl;
    }
    else {
        cout << "p1 == p2" << endl;
    }
}
int main() {
    test01();
    system("pause");
    return 0;
}
```

4.5.6 函数调用运算符重载

- 函数调用运算符 () 也可以重载
- 由于重载后的使用方式非常像函数的调用，因此称为仿函数
- 仿函数没有固定的写法，非常灵活

4.6 继承

继承是面向对象的三大特性之一

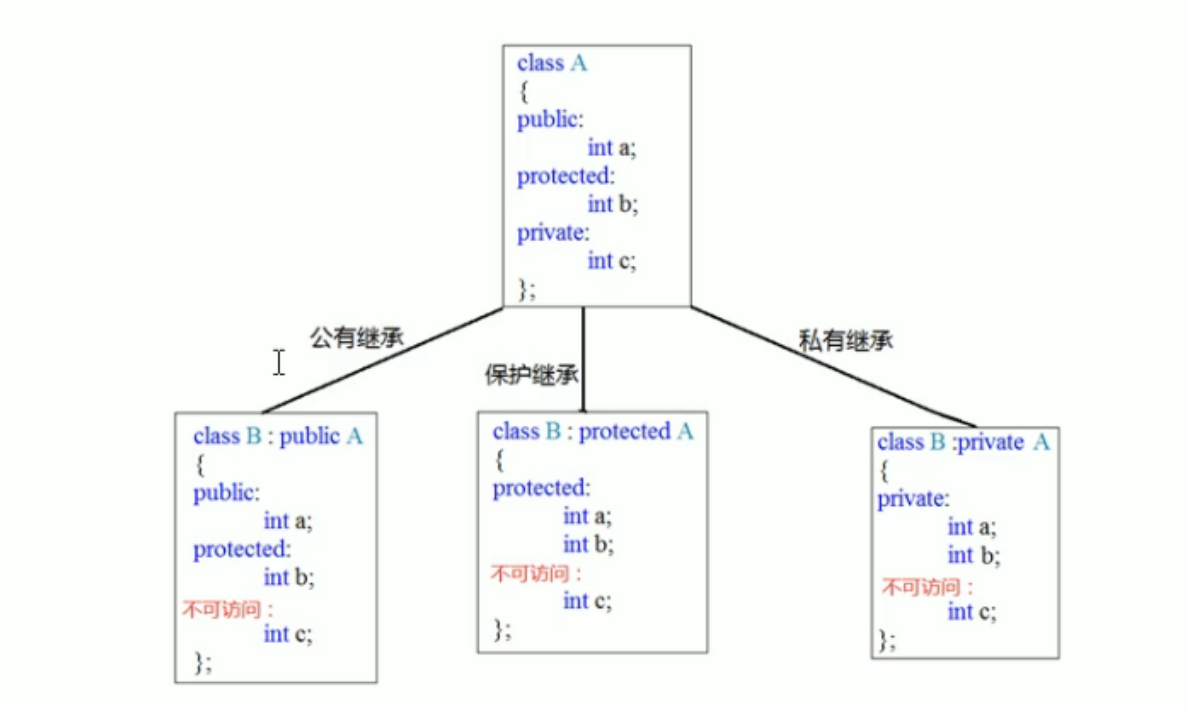
4.6.1 继承的基本语法

- 优点：减少重复代码
- 语法：

```
class 子类：继承方式 父类
 //子类（派生类），父类（基类）
```

4.6.2 继承方式

- 公共继承
- 保护继承
- 私有继承



4.6.3 继承中的对象模型

```cpp
#include <iostream>
using namespace std;
#include <string>
//继承中的对象模型
class Base {
public:
    int m_A;
protected:
    int m_B;
private:
    int m_C;
};
class Son :public Base {
public:
    int m_D;
};
void test01() {
    //父类中所有非静态成员属性都会被子类继承下去
    //父类中的私有属性被编译器隐藏，因此无法访问
    cout << "sizeof(Son) = " << sizeof(Son) << endl;
}
int main() {
    test01();
    system("pause");
    return 0;
}
```

- 利用开发人员命令工具查看对象模型
  - 跳转盘符
    - 查看所有文件 dir
  - 跳转文件路径 cd
  - 查看命名 cl /d1 reportSingleClassLayout类名 文件名

4.6.4 继承中构造和析构顺序

子类继承父类后，创建子类时会调用父类的构造函数

```
#include <iostream>
using namespace std;
#include <string>
//继承中的对象模型
class Base {
public:
    Base() {
        cout << "Base构造函数调用" << endl;
    }
    ~Base() {
        cout << "Base析构函数调用" << endl;
    }
};
class Son :public Base {
public:
    Son() {
        cout << "Son构造函数调用" << endl;
    }
    ~Son() {
        cout << "Son析构函数调用" << endl;
    }
};
void test01() {
    //创建子类对象时，先构造父类再构造子类
    //析构顺序与构造顺序相反
    Son s1;
}
int main() {
    test01();
    system("pause");
    return 0;
}
```

4.6.5继承中同名成员处理

子类和父类出现同名的成员，通过子类对象

- 访问子类同名函数，直接访问
- 访问父类同名函数，加作用域

```
#include <iostream>
using namespace std;
#include <string>
//继承中同名成员处理
class Base {
public:
    Base() {
        m_A = 100;
    }
    void func() {
        cout << "Base::func()调用" << endl;
    }
    void func(int a) {
```

```
        cout << "Base::func(int a)调用" << endl;
    }
    int m_A;
};
class Son :public Base {
public:
    Son() {
        m_A = 200;
    }
    void func() {
        cout << "Son::func()调用" << endl;
    }
    int m_A;
};
void test01() {
    Son s;
    //子类访问子类，直接访问
    cout << "Son::m_A = " << s.m_A << endl;
    s.func();
    //子类访问父类同名成员，加作用域
    cout << "Base::m_A = " << s.Base::m_A << endl;
    s.Base::func();

    //如果子类中出现与父类同名的成员，隐藏父类中所有的同名成员
    //访问加作用域
    s.Base::func(100);
}
int main() {
    test01();
    system("pause");
    return 0;
}
```

4.6.6 继承同名静态成员处理

静态成员和非静态成员出现同名，处理方式一致

- 访问子类同名函数，直接访问
- 访问父类同名函数，加作用域

```
#include <iostream>
using namespace std;
#include <string>
//继承同名静态成员处理
class Base {
public:
    static int m_A;
    static void func() {
        cout << "Base::func()调用" << endl;
    }
    static void func(int a) {
        cout << "Base::func(int a)调用" << endl;
    }
};
int Base::m_A = 100;
class Son :public Base {
public:
```

```
    static int m_A;
    static void func() {
        cout << "Son::func()调用" << endl;
    }

};
int Son::m_A = 200;
void test01() {
    //通过对象访问
    Son s;
    cout << "通过对象访问：" << endl;
    cout << "Son::m_A = " << s.m_A << endl;
    cout << "Base::m_A = " << s.Base::m_A << endl;
    s.func();
    s.Base::func();
    //通过类名
    cout << "通过类名访问：" << endl;
    cout << "Son::m_A = " << Son::m_A << endl;
    //cout << "Base::m_A = " << Base::m_A << endl;
    cout << "Base::m_A = " << Son::Base::m_A << endl;//第一个双冒号表示通过类名方式访问
    Son::func();
    Son::Base::func();
    Son::Base::func(100);
}
int main() {
    test01();
    system("pause");
    return 0;
}
```

4.6.7 多继承语法

C++允许一个类继承多个类

语法：

```
class 子类: 继承方式 父类1，继承方式 父类2.....
```

多继承中会引发父类中有同名成员出现，需要加作用域区分

4.6.8 菱形继承

概念（钻石继承）：

- 两个派生类继承同一个基类
- 某个类同时继承这两个派生类

问题：

- 使用数据时会产生二义性
- 资源浪费（相同的数据继承了两份）

```
#include <iostream>
using namespace std;
#include <string>
//菱形继承

```

```cpp
//动物类-->虚基类
class Animal{
public:
    int m_Age;
};
//利用虚继承解决菱形继承的问题
//羊类
class Sheep:virtual public Animal{};
//驼类
class Camel:virtual public Animal{};
//羊驼类
class Alpaca :public Sheep, public Camel {};
void test01() {
    Alpaca al;
    al.Sheep::m_Age = 18;
    al.Camel::m_Age = 28;
    cout << "al.Sheep::m_Age = " << al.Sheep::m_Age << endl;
    cout << "al.Camel::m_Age = " << al.Camel::m_Age << endl;
    cout << "al.m_Age = " << al.m_Age << endl;
}
int main() {
    test01();
    system("pause");
    return 0;
}
```

```
class Alpaca    size(12):
        +---
 0      | +--- (base class Sheep)
 0      | | {vbptr}
        | +---
 4      | +--- (base class Camel)
 4      | | {vbptr}
        | +---
        +---
        +--- (virtual base Animal)
 8      | m_Age
        +---

Alpaca::$vbtable@Sheep@:
 0      | 0
 1      | 8 (Alpacad(Sheep+0)Animal)

Alpaca::$vbtable@Camel@:
 0      | 0
 1      | 4 (Alpacad(Camel+0)Animal)
vbi:       class  offset o.vbptr  o.vbte fVtorDisp
          Animal       8       0       4 0
```

4.7 多态

4.7.1 多态的基本概念

多态分为两类：

- 静态多态：函数重载和运算符重载属于静态多态，复用函数名
- 动态多态：派生类和虚函数实现运行时多态

静态多态和动态多态的区别：

- 静态多态函数地址早绑定，编译阶段确定函数地址
- 动态多态函数地址晚绑定，运行阶段确定函数地址

```cpp
#include <iostream>
using namespace std;
#include <string>
//多态
class Animal {
public:
    //虚函数,函数地址晚绑定
    virtual void Speak() {
        cout << "动物在说话" << endl;
    }
};
class Cat :public Animal {
public:
    void Speak() {
        cout << "小猫在说话" << endl;
    }
};
class Dog :public Animal {
public:
    void Speak() {
        cout << "小狗在说话" << endl;
    }
};
//动态多态满足条件
//1、继承关系
//2、子类重写父类的虚函数
//动态多态的使用
//父类的指针或者引用指向子类对象
void doSpeak(Animal& animal) {
    animal.Speak();
}
void test01() {
    Cat cat;
    doSpeak(cat);//Animal& animal = cat;
    Dog dog;
    doSpeak(dog);
}
int main() {
    test01();
    system("pause");
    return 0;
}
```

```
class Animal    size(4):
        +---
 0      | {vfptr}
        +---

Animal::$vftable@:
        | &Animal_meta
        |  0
 0      | &Animal::Speak

Animal::Speak this adjustor: 0
```

```
class _s__CatchableType size(28):
        +---
 0      | properties
 4      | pType
 8      | _PMD thisDisplacement
20      | sizeOrOffset
24      | copyFunction
        +---

class _s__CatchableTypeArray    size(4):
        +---
 0      | nCatchableTypes
 4      | arrayOfCatchableTypes
        +---

class Cat       size(4):
        +---
 0      | +--- (base class Animal)
 0      | | {vfptr}
        | +---
        +---

Cat::$vftable@:
        | &Cat_meta
        |  0
 0      | &Cat::Speak
```

4.7.2 多态案例——计算器类

多态的优点：

- 代码组织结构清晰
- 可读性强
- 利于前期和后期的扩展以及维护

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

//多态写法
//开发中：开闭原则（对扩展开放，对修改封闭）
//计算器基类
class AbstractCalculator {
public:
    virtual int getResult() {
        return 0;
    }

    int m_Num1;
    int m_Num2;
};

//加法计算器类
class AddCalculator :public AbstractCalculator {
public:
    int getResult() {
        return m_Num1 + m_Num2;
    }
};
//减法计算器类
class SubCalculator :public AbstractCalculator {
public:
    int getResult() {
        return m_Num1 - m_Num2;
    }
};
//乘法计算器类
class MulCalculator :public AbstractCalculator {
public:
    int getResult() {
        return m_Num1 * m_Num2;
    }
};

void test01() {
    //多态使用条件：父类的指针或者引用指向子类对象
    //加法
    AbstractCalculator* abs = new AddCalculator;
    abs->m_Num1 = 100;
    abs->m_Num2 = 10;
    cout << abs->m_Num1 << "+" << abs->m_Num2 << "=" << abs->getResult() <<
endl;
    delete abs;
    //减法
    abs = new SubCalculator;
```

```
    abs->m_Num1 = 100;
    abs->m_Num2 = 10;
    cout << abs->m_Num1 << "-" << abs->m_Num2 << "=" << abs->getResult() << endl;
    delete abs;
    //乘法
    abs = new MulCalculator;
    abs->m_Num1 = 100;
    abs->m_Num2 = 10;
    cout << abs->m_Num1 << "*" << abs->m_Num2 << "=" << abs->getResult() << endl;
    delete abs;
    abs = NULL;
}
int main() {
    test01();

    system("pause");
    return 0;
}
```

4.7.3 纯虚函数和抽象类

父类中的虚函数的实现是毫无意义的，可以将虚函数改为**纯虚函数**

纯虚函数语法：

```
virtual 返回值类型 函数名(参数列表) = 0;
```

当类中有了纯虚函数，这个类也称为**抽象类**

抽象类特点：

- 无法实例化对象
- 子类必须重写抽象类中的纯虚函数，否则也属于抽象类

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

//纯虚函数和抽象类
class Base {
public:
    virtual void func() = 0;//纯虚函数
};
class Son :public Base {
public:
    //子类必须重写父类中的纯虚函数，否则也属于抽象类
    void func() {
        cout << "func()调用" << endl;
    }
};
void test01() {
    //Base b;   抽象类无法实例化对象
    //new Base; 抽象类无法实例化对象
    Base* base = new Son;
    base->func();
    delete base;
```

```
}
int main() {
    test01();

    system("pause");
    return 0;
}
```

4.7.4 多态案例二——制作饮品

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

//制作饮品
class AbstractDringking {
public:
    virtual void Boil() = 0;//煮水
    virtual void Brew() = 0;//冲泡
    virtual void PourInCup() = 0;//倒入杯中
    virtual void PutSomething() = 0;//加入辅料
    //制作饮品
    void MakeDrink() {
        Boil();
        Brew();
        PourInCup();
        PutSomething();
    }
};
class Coffee :public AbstractDringking {
    void Boil() {
        cout << "煮水" << endl;
    }
    void Brew() {
        cout << "冲泡咖啡" << endl;
    }
    void PourInCup() {
        cout << "倒入杯中" << endl;
    }
    void PutSomething() {
        cout << "加入牛奶和糖" << endl;
    }
};
class LenmonTea :public AbstractDringking {
    void Boil() {
        cout << "煮水" << endl;
    }
    void Brew() {
        cout << "冲泡茶叶" << endl;
    }
    void PourInCup() {
        cout << "倒入杯中" << endl;
    }
    void PutSomething() {
        cout << "加入柠檬" << endl;
    }
};
void doWork(AbstractDringking* abs) {
```

```
    abs->MakeDrink();
    delete abs;
}
void test01() {
    cout << "制作咖啡" << endl;
    doWork(new Coffee);
    cout << endl << "制作柠檬茶" << endl;
    doWork(new LenmonTea);
}
int main() {
    test01();

    system("pause");
    return 0;
}
```

4.7.5 虚析构和纯虚析构

多态使用时，如果子类中有属性开辟在堆区，那么父类指针在释放是无法调用到子类的析构代码——解决方式：将父类中的析构函数改为**虚析构**和**纯虚析构**

虚析构和纯虚析构共性：

- 可以解决父类指针释放子类对象
- 都需要有具体的函数实现

虚析构和纯虚析构区别：

- 如果是纯虚析构，该类属于抽象类，无法实例化对象

```
//虚析构语法：
virtual ~类名(){}
//纯虚析构语法
virtual ~类名() = 0;
类名::~类名(){}
```

```
#include <iostream>
using namespace std;
#include <string>
//虚析构和纯虚析构
class Animal {
public:
    Animal() {
        cout << "Animal构造函数调用" << endl;
    }
    virtual void speak() = 0;

    //纯虚析构---需要声明也需要实现
    //有了纯虚析构，这个类也属于抽象类
    virtual ~Animal() = 0;
    //虚析构--解决父类指针释放子类对象不干净的问题
    /*virtual ~Animal() {
        cout << "Animal析构函数调用" << endl;
    }*/
};
Animal::~Animal() {
    cout << "Animal纯虚析构函数调用" << endl;
```

```
}
class Cat :public Animal {
public:
    Cat(string name) {
        cout << "Cat构造函数调用" << endl;
        m_Name = new string(name);
    }
    virtual void speak() {
        cout << *m_Name << "小猫在说话" << endl;
    }
    ~Cat() {
        cout << "Cat析构函数调用" << endl;
        if (m_Name != NULL) {
            delete m_Name;
            m_Name = NULL;
        }
    }
    string* m_Name;
};

void test01() {
    Animal* animal = new Cat("Tom");
    animal->speak();
    delete animal;
}
int main() {
    test01();

    system("pause");
    return 0;
}
```

4.7.6 多态案例三——组装电脑

```
#include <iostream>
#include <string>

using namespace std;

//抽象的CPU类
class CPU {
public:
    //抽象的计算函数
    virtual void calculate() = 0;
};
//抽象的显卡类
class VideoCard {
public:
    //抽象的显示函数
    virtual void display() = 0;
};
//抽象的内存条类
class Memory {
public:
    //抽象的存储函数
    virtual void storage() = 0;
};
```

```
24
25  //电脑类
26  class Computer {
27  public:
28      Computer(CPU* cpu, VideoCard* vc, Memory* mem) {
29          m_cpu = cpu;
30          m_vc = vc;
31          m_mem = mem;
32      }
33      //工作函数
34      void work() {
35          m_cpu->calculate();
36          m_vc->display();
37          m_mem->storage();
38      }
39      ~Computer() {
40          if (m_cpu != NULL) {
41              delete m_cpu;
42              m_cpu = NULL;
43          }
44          if (m_vc != NULL) {
45              delete m_vc;
46              m_vc = NULL;
47          }
48          if (m_mem != NULL) {
49              delete m_mem;
50              m_mem = NULL;
51          }
52      }
53  private:
54      //零件指针
55      CPU* m_cpu;
56      VideoCard* m_vc;
57      Memory* m_mem;
58  };
59  //具体厂商
60  //Intel厂商
61  class IntelCpu :public CPU {
62  public:
63      void calculate() {
64          cout << "Intel的CPU开始计算了" << endl;
65      }
66  };
67  class IntelVedioCard :public VideoCard {
68  public:
69      void display() {
70          cout << "Intel的显卡开始显示了" << endl;
71      }
72  };
73  class IntelMemory :public Memory {
74  public:
75      void storage() {
76          cout << "Intel的内存条开始存储了" << endl;
77      }
78  };
79  //Lenovo厂商
80  class LenovoCpu :public CPU {
81  public:
```

```
    void calculate() {
        cout << "Lenovo的CPU开始计算了" << endl;
    }
};
class LenovoVedioCard :public VideoCard {
public:
    void display() {
        cout << "Lenovo的显卡开始显示了" << endl;
    }
};
class LenovoMemory :public Memory {
public:
    void storage() {
        cout << "Lenovo的内存条开始存储了" << endl;
    }
};
void test01() {
    //第一台电脑
    CPU* intelCpu = new IntelCpu;
    VideoCard* intelVedioCard = new IntelVedioCard;
    Memory* intelMemory = new IntelMemory;

    //创建第一台电脑
    Computer* computer1 = new Computer(intelCpu, intelVedioCard,
intelMemory);
    computer1->work();
    delete computer1;

    //创建第二台电脑
    Computer* computer2 = new Computer(new LenovoCpu, new LenovoVedioCard,
new LenovoMemory);
    computer2->work();
    delete computer2;
}
int main() {
    test01();

    system("pause");
    return 0;
}
```

5 文件操作

文件可以将数据永久化

文件类型：

- 文本文件：文件以文本的ASCLL码形式存储在计算机中
- 二进制文件：文件以文本的二进制形式存储在计算级中

操作文件：

- ofstrem 写操作
- ifstrem 读操作
- fstream 读写操作

5.1 文本文件

5.1.1 写文件

步骤：

- 包含头文件
- 创建流对象
- 打开文件
- 写数据
- 关闭文件

```
#include <fstream>
ofstream ofs;
ofs.open("文件路径",打开方式);
ofs << "写入的数据";
ofs.close();
```

打开方式	解释
ios::in	为读文件而打开文件
ios::out	为写文件而打开文件
ios::ate	初始位置：文件尾
ios::app	追加方式写文件
ios::trunc	如果文件存在先删除，再创建
ios::binary	二进制方式

- 文件打开方式可以配合使用，使用 | 连接

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

//写文件
void test01() {
    ofstream ofs;//创建文件流
    ofs.open("test.txt", ios::out);//打开文件
    ofs << "姓名：张三" << endl
        << "年龄：18" << endl
        << "性别：男" << endl;//写入数据
    ofs.close();//关闭文件
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.1.2 读文件

步骤:

- 包含头文件
- 创建流对象
- 打开文件并判断文件是否打打开成功
- 读数据（四种方式读取）
- 关闭文件

```cpp
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

//读文件
void test01() {
    ifstream ifs;//创建流对象
    ifs.open("E:/系统默认/桌面/test.txt", ios::in);//打开文件并判断是否打开成功
    if (!ifs.is_open()) {
        cout << "文件打开失败" << endl;
        return;
    }
    //读文件

    //第一种
    char buffer[1024] = { 0 };
    while (ifs >> buffer) {
        cout << buffer << endl;
    }

    //第二种
    char buffer[1024] = { 0 };
    while (ifs.getline(buffer, sizeof(buffer))) {
        cout << buffer << endl;
    }

    //第三种
    string buffer;
    while (getline(ifs, buffer)) {
        cout << buffer << endl;
    }

    //第四种
    char c;
    while ((c = ifs.get()) != EOF) {
        cout << c;
    }
    ifs.close();
}

int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.2 二进制文件

5.2.1 写文件

二进制方式写文件主要利用流对象调用成员函数write

函数原型：ostream& write(const char * buffer,int len);

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

//二进制读文件
class Person {
public:
    char m_Name[64];
    int m_Age;
};
void test01() {
    ofstream ofs("person.txt", ios::out | ios::binary);//创建流对象
    //ofs.open("person.txt", ios::out | ios::binary);
    Person p = { "Tom",18 };
    ofs.write((const char*)&p, sizeof(Person));
    ofs.close();
}

int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.2.2 读文件

二进制方式写文件主要利用流对象调用成员函数read

函数原型：istream& read(char * buffer,int len);

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

//二进制读文件
class Person {
public:
    char m_Name[64];
    int m_Age;
};
void test01() {
    ifstream ifs;
    ifs.open("person.txt", ios::in | ios::binary);
    if (!ifs.is_open()) {
        cout << "文件打开失败" << endl;
        return;
```

```
    }
    Person p;
    ifs.read((char*)&p, sizeof(Person));
    cout << "姓名: " << p.m_Name << endl;
    cout << "年龄: " << p.m_Age << endl;
    ifs.close();
}

int main()
{
    test01();
    system("pause");
    return 0;
}
```