

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

```
1  #include <iostream>
2  using namespace std;
3
4  //全局变量
5  int g_a = 10;
6  int g_b = 10;
7  const int c_g_a = 10;
8  const int c_g_b = 10;
9  int main() {
10     //全局区
11
12     //普通局部变量
13     int a = 10;
14     int b = 10;
15     cout << "局部变量a的地址: " << &a << endl;
16     cout << "局部变量b的地址: " << &b << endl;
17
18     cout << "全局变量g_a的地址: " << &g_a << endl;
19     cout << "全局变量g_b的地址: " << &g_b << endl;
20
21     //静态变量
22     static int s_a = 10;
```

```

23     static int s_b = 10;
24     cout << "静态变量s_a的地址: " << &s_a << endl;
25     cout << "静态变量s_b的地址: " << &s_b << endl;
26
27     //常量
28     //字符串常量
29     cout << "字符串常量的地址: " << &"HelloWorld" << endl;
30     //const修饰的变量
31     //const修饰的全局变量
32     cout << "全局常量c_g_a的地址: " << &c_g_a << endl;
33     cout << "全局常量c_g_b的地址: " << &c_g_b << endl;
34     //const 修饰的局部变量
35     const int c_l_a = 10;
36     const int c_l_b = 10;
37     cout << "局部常量c_l_a的地址: " << &c_l_a << endl;
38     cout << "局部常量c_l_b的地址: " << &c_l_b << endl;
39     system("pause");
40     return 0;
41 }

```

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

```

1  #include <iostream>
2  using namespace std;
3
4  //栈区的注意事项--不要返回局部变量的地址
5  //栈区的数据由编译器开辟释放
6
7  int* func() {
8      int a = 10; //局部变量
9      return &a;
10 }
11 int main() {
12     cout << *(func()) << endl;
13     int* p = func(); //vs2022 分号结束 内存被释放
14     cout << " *p = " << *p << endl;
15     system("pause");
16     return 0;
17 }

```

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

```

1  #include <iostream>
2  using namespace std;
3
4  //在堆区开辟数据
5  int* func() {
6      //利用new关键字，可以将数据开辟到堆区
7      //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
8      int* p = new int(10);
9      return p;
10 }
11 int main() {
12     int* p = func();
13     cout << "*p = " << *p << endl;
14     cout << "*p = " << *p << endl;
15     system("pause");
16     return 0;
17 }

```

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

```

1  #include <iostream>
2  using namespace std;
3
4  //new与delete
5  int* func() {
6      //在堆区创建一个整形数据
7      int* p = new int(10);
8      return p;
9  }
10 void test01() {
11     int* p = func();
12     cout << *p << endl;
13     //释放堆区内存
14     delete p;
15     //cout << *p << endl; //内存已被释放，再次访问属于非法操作
16 }
17 void test02() {
18     //堆区开辟数组
19     //创建10个整形数据的数组
20     int* arr = new int[10]; //返回首地址
21     for (int i = 0; i < 10; i++) {
22         arr[i] = i + 100;
23     }
24     for (int i = 0; i < 10; i++) {
25         cout << arr[i] << "\t" << endl;
26     }
27     //释放堆区数组
28     delete[] arr;
29 }

```

```

30 int main() {
31     test01();
32     test02();
33     system("pause");
34     return 0;
35 }

```

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

```

1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      //引用基本语法：数据类型 &别名 = 原名
6      int a = 10;
7      //创建引用
8      int& b = a;
9      cout << "a = " << a << endl;
10     cout << "b = " << b << endl;
11
12     b = 100;
13     cout << "a = " << a << endl;
14     cout << "b = " << b << endl;
15
16     system("pause");
17     return 0;
18 }

```

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

```

1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int a = 10;
6      int& b = a;
7      //引用的注意事项
8      //1、引用必须初始化
9      //int& c;    错误
10     //2、引用在初始化后，不可以改变
11     int c = 20;
12     //int& b = c;    错误，重定义，多次初始化
13     system("pause");
14     return 0;
15 }

```

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

```
1  #include <iostream>
2  using namespace std;
3
4  //交换函数
5  //1、值传递
6  void Swap01(int a, int b) {
7      int temp = a;
8      a = b;
9      b = temp;
10 }
11 //2、地址传递
12 void Swap02(int* a, int* b) {
13     int temp = *a;
14     *a = *b;
15     *b = temp;
16 }
17 //3、引用传递
18 void Swap03(int& a, int& b) {
19     int temp = a;
20     a = b;
21     b = temp;
22 }
23 int main() {
24     int a = 10;
25     int b = 20;
26     Swap01(a, b); //值传递，形参不会修饰实参
27     cout << "a = " << a << endl;
28     cout << "b = " << b << endl;
29     Swap02(&a, &b); //地址传递，形参会修饰实参
30     cout << "a = " << a << endl;
31     cout << "b = " << b << endl;
32     Swap03(a, b); //引用传递，形参会修饰实参
33     cout << "a = " << a << endl;
34     cout << "b = " << b << endl;
35     system("pause");
36     return 0;
37 }
```

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

```
1  #include <iostream>
2  using namespace std;
3
4  //引用做函数返回值
5  //1、不要返回局部变量的引用
```

```

6  int& test01() {
7      int a = 10;
8      return a;
9  }
10 int& test02() {
11     static int a = 10; //静态变量，存放在全局区
12     return a;
13 }
14 int main() {
15     //int& ref = test01();
16     //cout << "ref = " << ref << endl; //结果错误，因为a的内存已经释放
17     int& ref2 = test02();
18     cout << "ref2 = " << ref2 << endl;
19     //2、函数的调用可以作为左值
20     test02() = 1000; //test02()返回a的引用，ref2是a的别名
21     cout << "ref2 = " << ref2 << endl;
22
23     system("pause");
24     return 0;
25 }

```

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

```

1  //发现是引用，转换成 int* const ref = &a;
2  void func(int& ref){
3      ref = 100; //ref是引用，转换成*ref = 100
4  }
5  int main(){
6      int a = 10;
7      //自动转换成int* const ref = &a; 指针常量是指针指向不可改
8      int& ref = a;
9      ref = 20; //内部发现ref是引用，自动转换成*ref = 20;
10     return 0;
11 }

```

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

```

1  #include <iostream>
2  using namespace std;
3  //打印数据函数
4  void ShowValue(const int& val) {
5      //val = 1000; 错误
6      cout << "val = " << val << endl;
7  }
8  int main() {
9      //常量引用
10     //使用场景，用来修饰形参，防止误操作
11     int a0 = 10;
12     //加上const之后，编译器将代码修改，int temp = 10; const int &ref = temp;
13     const int& ref = 10; //引用必须引一块合法的内存空间

```

```

14 //ref = 20;加入const修饰后变成只读
15 int a = 100;
16 ShowValue(a);
17 cout << "a = " << a << endl;
18
19 system("pause");
20 return 0;
21 }

```

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)

```

1  #include <iostream>
2  using namespace std;
3
4  //函数的默认参数
5  // 语法：返回值类型 函数名（形参 = 默认值）
6  //如果传入数据，掩盖原有的默认值
7
8  int func(int a,int b = 20,int c = 30) {
9      return a + b + c;
10 }
11 //注意事项：
12 //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
13 //int func2(int a, int b = 10, int c, int d);//默认实参不在形参列表结尾
14
15 //2、如果函数的声明有默认参数，函数实现不能有默认参数
16 //声明和实现只能有一个有默认参数
17 int func2(int a=10, int b=20);
18 //int func2(int a=10, int b=20) {重定义默认参数
19 //  return a + b;
20 //}
21 int func2(int a, int b) {
22     return a + b;
23 }
24 int main() {
25     cout << func(10) << endl;
26     cout << func2() << endl;
27     system("pause");
28     return 0;
29 }

```

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）{ }

```

1  #include <iostream>
2  using namespace std;
3

```

```

4 //占位参数
5 //返回值类型 函数名(数据类型){}
6 //占位参数可以有默认参数
7 void func(int a,int = 10) {
8     cout << "func()调用" << endl;
9 }
10 int main()
11 {
12     func(10,10);
13     func(10);
14     system("pause");
15     return 0;
16 }

```

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

```

1 #include <iostream>
2 using namespace std;
3
4 //函数重载条件
5 /*
6     1、同一作用域
7     2、函数名称相同
8     3、函数参数类型不同或者个数不同或者顺序不同
9 */
10 void func() {
11     cout << "func()的调用" << endl;
12 }
13 void func(int a) {
14     cout << "func(int)的调用" << endl;
15 }
16 void func(int a, double b) {
17     cout << "func(int,double)的调用" << endl;
18 }
19 void func(double b,int a) {
20     cout << "func(double,int)的调用" << endl;
21 }
22 //注意事项：函数的返回值不可以作为函数重载的条件
23 //int func(double b, int a) {
24 //    cout << "func(double,int)的调用" << endl;
25 //    return a;
26 //}
27 int main()
28 {
29     func();

```



```

30     func(10);
31     func(10, 10.0);
32     func(10.0, 10);
33     system("pause");
34     return 0;
35 }

```

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

```

1  #include <iostream>
2  using namespace std;
3
4  //函数重载的注意事项
5  //1、引用作为重载的条件
6  void func(int &a) {
7      cout << "func(int &a)调用" << endl;
8  }
9  void func(const int& a) {
10     cout << "func(const int &a)调用" << endl;
11 }
12 //2、函数重载碰到默认参数
13 void func2(int a,int b = 10) {
14     cout << "func2(int a,int b = 10)调用" << endl;
15 }
16 void func2(int a) {
17     cout << "func2(int a)调用" << endl;
18 }
19 int main()
20 {
21     int a = 10;
22     func(a); //a可读可写, 调用无const的版本
23     func(10); //10只读,int &a = 10,不合法;const int &a = 10,合法;
24     // func2(10);当函数重载碰到默认参数, 出现二义性, 尽量避免
25     func2(10, 20);
26
27     system("pause");
28     return 0;
29 }

```

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};

```
1  #include <iostream>
2  using namespace std;
3
4  const double PI = 3.14;
5  //设计一个圆类，求圆的周长
6  class Circle {
7  //访问权限
8  public:
9      //属性
10     int m_r;//半径
11     //行为
12     double calculateZC() { //获取圆的周长
13         return 2 * PI * m_r;
14     }
15 };
16 int main()
17 {
18     Circle c1;//创建圆对象
19     c1.m_r = 10;
20     cout << "圆的周长为: " << c1.calculateZC() << endl;
21     system("pause");
22     return 0;
23 }
```

- 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

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  //学生类
5  class Student {
6  public:
7      //类中的属性和行为同一称为成员
8      //属性：成员属性/成员变量
9      //行为：成员函数/成员方法
10     void SetName(string name) {
11         m_name = name;
12     }
13     void SetId(int id) {
14         m_id = id;
15     }
16     void ShowStudent() {
```

```

17         cout << "姓名: " << m_name << "\t学号: " << m_id << endl;
18     }
19     string m_name;
20     int m_id;
21
22 };
23 int main()
24 {
25     Student stu1;
26     string name;
27     int id;
28     cout << "请输入姓名: ";
29     cin >> name;
30     cout << endl;
31     cout << "请输入学号: ";
32     cin >> id;
33     cout << endl;
34     stu1.SetName(name);
35     stu1.SetId(id);
36     stu1.ShowStudent();
37     system("pause");
38     return 0;
39 }

```

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

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  //访问权限
6  //公共权限 public      成员类内类外皆可访问
7  //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
8  //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
9  class Person {
10 public:
11     string m_Name;//姓名
12 protected:
13     string m_Car;//汽车
14 private:
15     int m_Password;//银行卡密码
16
17 public:
18     //类内访问
19     void func() {
20         m_Name = "Tom";
21         m_Car = "BMW";
22         m_Password = 123;
23     }
24 };

```

```

25  int main()
26  {
27      Person p1; //实例化具体对象
28      p1.m_Name = "Jerry";
29      p1.func();
30      //类外不可访问保护权限和私有权限内容
31      //p1.m_Car = "拖拉机";
32      //p1.m_Password = 456;
33      system("pause");
34      return 0;
35  }

```

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访问权限不同

区别：

- struct默认权限为公共
- class默认权限为私有

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class C1 {
6      int m_A; //默认权限是私有
7  };
8  struct C2 {
9      int m_A; //默认权限是公共
10 };
11 int main()
12 {
13     C1 c1;
14     //c1.m_A = 10; 私有权限类外不可访问
15     C2 c2;
16     c2.m_A = 10;
17     system("pause");
18     return 0;
19 }

```

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

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  //成员属性设置为私有
6  class Person {
7  public:
8      //提供接口
9      void SetName(string name) {
10         m_Name = name;

```

```

11     }
12     string GetName() {
13         return m_Name;
14     }
15     void SetAge(int age) {
16         //判断数据的有效性
17         if (!(age > 0 && age < 150)) {
18             cout << "非法输入! " << endl;
19             return;
20         }
21         m_Age = age;
22     }
23     int GetAge() {
24         return m_Age;
25     }
26     void SetLover(string lover) {
27         m_Lover = lover;
28     }
29 private:
30     string m_Name; //姓名---可读可写
31     int m_Age; //年龄---只读
32     string m_Lover; //情人---只写
33 };
34 int main()
35 {
36     Person p;
37     p.SetName("张三");
38     cout << "姓名: " << p.GetName() << endl;
39     p.SetAge(75);
40     cout << "年龄: " << p.GetAge() << endl;
41     p.SetLover("李四");
42     system("pause");
43     return 0;
44 }

```

- 封装案例一：设计立方体类

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  //设计立方体类
6  //求出立方体的面积和体积
7  //分别用全局函数和成员函数来判断两个立方体是否相等
8  class Cube {
9  public:
10     //设置长
11     void SetL(double l) {
12         m_L = l;
13     }
14     //获取长
15     double GetL() {
16         return m_L;
17     }
18     //设置宽
19     void SetW(double w) {
20         m_W = w;

```

```

21     }
22     //获取宽
23     double GetW() {
24         return m_W;
25     }
26     //设置高
27     void SetH(double h) {
28         m_H = h;
29     }
30     //获取高
31     double GetH() {
32         return m_H;
33     }
34     //获取立方体面积
35     double calculates() {
36         return 2 * (m_L * m_W + m_L * m_H + m_W * m_H);
37     }
38     //获取立方体面积
39     double calculatev() {
40         return m_L * m_W * m_H;
41     }
42     //成员函数判断
43     bool issameByClass(Cube& c) {
44         if (m_L == c.GetL() && m_W == c.GetW() && m_H == c.GetH()) {
45             return true;
46         }
47         return false;
48     }
49 private:
50     double m_L; //长
51     double m_W; //宽
52     double m_H; //高
53 };
54
55 //全局函数判断
56 bool issame(Cube& c1, Cube& c2) {
57     if (c1.GetL() == c2.GetL() && c1.GetW() == c2.GetW() && c1.GetH() ==
58         c2.GetH()) {
59         return true;
60     }
61     return false;
62 }
63 int main()
64 {
65     Cube c1; //实例化立方体对象
66     c1.SetL(10.0);
67     c1.SetW(10.0);
68     c1.SetH(10.0);
69     cout << "c1的面积为: " << c1.Calculates() << endl;
70     cout << "c1的体积为: " << c1.Calculatev() << endl;
71     Cube c2; //实例化立方体对象
72     c2.SetL(10.0);
73     c2.SetW(10.2);
74     c2.SetH(10.0);
75     //bool ret = issame(c1, c2);
76     bool ret = c1.issameByClass(c2);
77     if (ret) {
78         cout << "c1 == c2" << endl;

```

```

78     }
79     else {
80         cout << "c1 != c2" << endl;
81     }
82     system("pause");
83     return 0;
84 }

```

- 封装案例二：点和圆的关系

```

1  //main.cpp
2  #include <iostream>
3  #include <string>
4  using namespace std;
5  #include "point.h"
6  #include "circle.h"
7  //判断点和圆的关系
8  //判断
9  void isInCircle(Circle& c, Point& p) {
10     //计算两点间距离的平方
11     int distance = (c.GetCenter().GetX() - p.GetX()) * (c.GetCenter().GetX()
12 - p.GetX()) +
13     (c.GetCenter().GetY() - p.GetY()) * (c.GetCenter().GetY() -
14 p.GetY());
15     //计算半径的平方
16     int rDistance = c.GetR() * c.GetR();
17     //判断关系
18     if (distance == rDistance) {
19         cout << "点在圆上" << endl;
20     }
21     else if (distance > rDistance) {
22         cout << "点在圆外" << endl;
23     }
24     else {
25         cout << "点在圆内" << endl;
26     }
27 }
28 int main()
29 {
30     //实例化圆
31     Circle c;
32     c.SetR(10);
33     Point center;
34     center.SetX(10);
35     center.SetY(0);
36     c.SetCenter(center);
37     //实例化点
38     Point p;
39     p.SetX(10);
40     p.SetY(5);
41     //判断关系
42     isInCircle(c, p);
43     system("pause");
44     return 0;
45 }

```

```

1 //point.h
2 #pragma once
3 #include <iostream>
4 using namespace std;
5 //点类
6 class Point {
7 private:
8     int m_X;
9     int m_Y;
10 public:
11     //设置获取X
12     void SetX(int x);
13     int GetX();
14     //设置获取Y
15     void SetY(int y);
16     int GetY();
17 };

```

```

1 //point.cpp
2 #include "point.h"
3 //设置获取X
4 void Point::SetX(int x) {
5     m_X = x;
6 }
7 int Point::GetX() {
8     return m_X;
9 }
10 //设置获取Y
11 void Point::SetY(int y) {
12     m_Y = y;
13 }
14 int Point::GetY() {
15     return m_Y;
16 }

```

```

1 //circle.h
2 #pragma once
3 #include <iostream>
4 using namespace std;
5 #include "point.h"
6 //圆类
7 class Circle {
8 public:
9     //设置获取半径
10     void SetR(int r);
11     int GetR();
12     //设置获取圆心
13     void SetCenter(Point center);
14     Point GetCenter();
15 private:
16     int m_R;//半径
17     Point m_Center;
18 };

```

```

1 //circle.cpp

```



```

2  #include "circle.h"
3  //设置获取半径
4  void Circle::SetR(int r) {
5      m_R = r;
6  }
7  int Circle::GetR() {
8      return m_R;
9  }
10 //设置获取圆心
11 void Circle::SetCenter(Point center) {
12     m_Center = center;
13 }
14 Point Circle::GetCenter() {
15     return m_Center;
16 }

```

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

```

1  #include <iostream>
2  using namespace std;
3
4  //对象的初始化和清理
5  class Person {
6  public:
7      //1、构造函数
8      Person() {
9          cout << "Person() 构造函数调用" << endl;

```

```

10     }
11
12     //2、析构函数
13     ~Person() {
14         cout << "~Person()析构函数调用" << endl;
15     }
16 };
17 void test01() {
18     Person p;
19 }
20 int main()
21 {
22     test01();
23     //Person p;
24     system("pause");
25     return 0;
26 }

```

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

```

1  #include <iostream>
2  using namespace std;
3
4  //构造函数的分类及调用
5  //分类
6  //按参数分类：无参构造（默认构造）和有参构造
7  //按类型分类：普通构造 拷贝构造
8  class Person {
9  public:
10     //构造函数
11     Person() {
12         m_Age = 18;
13         cout << "Person的无参构造函数调用" << endl;
14     }
15     Person(int a) {
16         m_Age = a;
17         cout << "Person的有参构造函数调用" << endl;
18     }
19     //拷贝构造函数
20     Person(const Person& p) {
21         //将传入的类的属性拷贝
22         m_Age = p.m_Age;
23         cout << "Person的拷贝构造函数调用" << endl;
24     }
25     }
26     int m_Age;
27     //析构函数

```

```

28     ~Person() {
29         cout << "Person的析构函数调用" << endl;
30     }
31 };
32 //调用
33 void test01() {
34     //1、括号法
35     Person p1;           //默认构造函数的调用
36     Person p2(18);       //有参构造函数的调用
37     Person p3(p2);       //拷贝构造函数的调用
38     //注意事项:
39     //1、调用默认构造函数时不要加()
40     //Person p1();编译器认为这是函数声明, 不会认为在创建对象
41
42     //2、显示法
43     Person p4 = Person(18); //有参构造函数的调用
44     Person p5 = Person(p2); //拷贝构造函数的调用
45     //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
46     //注意事项:
47     //2、不要利用拷贝构造函数初始化匿名对象
48     //Person(p4);编译器认为Person(p4) === Person p4, 对象声明, 重定义
49
50     //3、隐式转换法
51     Person p6 = 10; //有参构造函数调用    Person p4 = Person(18);
52     Person p7 = p6; //拷贝构造函数调用
53 }
54 int main()
55 {
56     test01();
57     system("pause");
58     return 0;
59 }

```

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

```

1  #include <iostream>
2  using namespace std;
3
4  //拷贝构造函数的调用时机
5
6  class Person {
7  public:
8      Person() {
9          cout << "Person默认构造函数调用" << endl;
10     }
11     Person(int age) {
12         cout << "Person有参构造函数调用" << endl;
13         m_Age = age;
14     }
15     Person(const Person& p) {
16         cout << "Person拷贝构造函数的调用" << endl;

```

```

17         m_Age = p.m_Age;
18     }
19     ~Person() {
20         cout << "Person析构函数调用" << endl;
21     }
22     int m_Age;
23 };
24 //1、使用一个已经创建完毕的对象来创建一个新对象
25 void test01() {
26     Person p1(20);
27     Person p2(p1);
28     cout << "p2.m_Age = " << p2.m_Age << endl;
29 }
30 //2、值传递方式给函数参数传值
31 void dowork(Person p) {
32     //值传递的本质：拷贝一份副本数据
33 }
34 void test02() {
35     Person p;
36     dowork(p);
37 }
38 //3、值方式返回局部对象
39 Person dowork2() {
40     Person p1;
41     cout << "&p1 = " << &p1 << endl;
42     return p1; //返回一个拷贝的p1
43 }
44 void test03() {
45     Person p = dowork2();
46     cout << "&p = " << &p << endl;
47 }
48 int main()
49 {
50     test01();
51     test02();
52     test03();
53     system("pause");
54     return 0;
55 }

```

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

```

1 #include <iostream>
2 using namespace std;
3
4 //构造函数调用规则
5

```

```

6  class Person {
7  public:
8      //无参（默认）构造函数
9      Person() {
10         cout << "person无参构造函数调用" << endl;
11     }
12     //有参构造函数
13     Person(int age) {
14         cout << "Person有参构造函数调用" << endl;
15         m_Age = age;
16     }
17     //拷贝构造函数
18     Person(const Person& p) {
19         cout << "Person拷贝构造函数调用" << endl;
20         m_Age = p.m_Age;
21     }
22
23     ~Person() {
24         cout << "Person析构函数调用" << endl;
25     }
26     int m_Age;
27 };
28 void test01() {
29     Person p;
30     p.m_Age = 18;
31
32     Person p2(p);
33     cout << "p2.m_Age = " << p2.m_Age << endl;
34 }
35 void test02() {
36     //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
37     //如果用户定义拷贝构造函数，C++不再提供其他构造函数
38     Person p1(19);
39     Person p2(p1);
40 }
41
42 int main()
43 {
44     test01();
45     test02();
46     system("pause");
47     return 0;
48 }

```

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

```

1  #include <iostream>
2  using namespace std;
3
4  //深拷贝和浅拷贝
5  class Person {
6  public:
7      Person() {
8         cout << "Person的默认构造函数调用" << endl;

```

```

9      }
10     Person(int age,int height) {
11         m_Age = age;
12         m_Height = new int(height);
13         cout << "Person的有参构造函数调用" << endl;
14     }
15     Person(const Person& p) {
16         m_Height = new int(*p.m_Height);
17         m_Age = p.m_Age;
18         cout << "Person拷贝构造函数的调用" << endl;
19     }
20     ~Person() {
21         if (m_Height != NULL) {
22             delete m_Height;
23             m_Height = NULL;
24         }
25         cout << "Person的析构函数调用" << endl;
26     }
27     int m_Age;
28     int* m_Height;
29 };
30 void test01() {
31     Person p1(18,160);
32     cout << "p1.m_Age = " << p1.m_Age << endl;
33     cout << "*p1.m_Height = " << *p1.m_Height << endl;
34     //编译器提供的拷贝构造函数做的是浅拷贝的操作
35     //浅拷贝带来的问题是堆区内内存重复释放-----深拷贝解决
36     Person p2(p1);
37     cout << "p2.m_Age = " << p2.m_Age << endl;
38     cout << "*p1.m_Height = " << *p2.m_Height << endl;
39 }
40 }
41 int main()
42 {
43     test01();
44     system("pause");
45     return 0;
46 }

```

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),.....{};

```

1  #include <iostream>
2  using namespace std;
3
4  //初始化列表
5  class Person {
6  public:
7
8      //传统初始化操作
9      /*Person(int a,int b,int c) {
10         m_A = a;
11         m_B = b;
12         m_C = c;

```

```

13     */
14     //初始化列表
15     Person(int a,int b,int c) :m_A(a), m_B(b), m_C(c) {
16
17     }
18     int m_A;
19     int m_B;
20     int m_C;
21 };
22 void test01() {
23     Person p(10, 20, 30);
24     cout << "m_A = " << p.m_A << endl;
25     cout << "m_B = " << p.m_B << endl;
26     cout << "m_C = " << p.m_C << endl;
27 }
28 int main()
29 {
30     test01();
31     system("pause");
32     return 0;
33 }

```

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：

```

1 class A;
2 class B{
3     A a;//A为对象成员
4 }

```

```

1 #include <iostream>
2 using namespace std;
3 #include <string>
4
5 //类对象作为类成员
6 class Phone {
7 public:
8     Phone(string pName) {
9         cout << "Phone有参构造函数调用" << endl;
10        m_PName = pName;
11    }
12    ~Phone() {
13        cout << "Phone析构函数调用" << endl;
14    }
15    string m_PName;
16 };
17 class Person{
18 public:
19     //Phone m_Phone = pName;隐式转换法
20     Person(string name, string pName) :m_Name(name), m_Phone(pName) {
21         cout << "Person有参构造函数调用" << endl;
22     }
23     ~Person() {
24         cout << "Person析构函数调用" << endl;

```

```

25     }
26     string m_Name;
27     Phone m_Phone;
28 };
29 //当其他类对象作为本类成员，先构造其他类，在构造自身
30 //析构顺序与构造顺序相反
31 void test01() {
32     Person p("Tom", "iPhone");
33     cout << p.m_Name << " with " << p.m_Phone.m_PName << endl;
34 }
35 int main()
36 {
37     test01();
38     system("pause");
39     return 0;
40 }

```

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

```

1  #include<iostream>
2  using namespace std;
3
4  //静态成员变量
5  class Person {
6  public:
7      static int m_A; //类内声明
8      //静态成员拥有访问权限
9  private:
10     static int m_B;
11 };
12 int Person::m_A = 100; //类外初始化
13 int Person::m_B = 200;
14 void test01() {
15     Person p;
16     cout << "p.m_A = " << p.m_A << endl;
17     //cout << "p.m_B = " << p.m_B << endl; 类外不可访问
18     Person p2;
19     p2.m_A = 200;
20     cout << "p.m_A = " << p.m_A << endl;
21 }
22 void test02() {
23     //静态成员不属于某个对象上
24     //因此静态成员变量有两种访问方式
25
26     //1、通过对象访问
27     Person p;
28     cout << "p.m_A = " << p.m_A << endl;
29     //2、通过类名访问

```



```

30     cout << "Person::m_A = " << Person::m_A << endl;
31 }
32 int main()
33 {
34     //test01();
35     test02();
36     system("pause");
37     return 0;
38 }

```

- 静态成员函数

- 所有对象共享一个函数
- 静态成员函数只能访问静态成员变量

```

1  #include<iostream>
2  using namespace std;
3
4  //静态成员函数
5  class Person {
6  public:
7      static void func() {
8          m_A = 100; //静态成员函数可以访问静态成员变量
9          //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
10         cout << "static void func()调用" << endl;
11     }
12     static int m_A;
13     int m_B;
14     //静态成员函数拥有访问权限
15 private:
16     static void func2() {
17         cout << "static void func2()调用" << endl;
18     }
19 };
20 int Person::m_A = 0;
21 void test01() {
22     //访问静态成员函数
23     //1、通过对象访问
24     Person p;
25     p.func();
26     //2、通过类名访问
27     Person::func();
28     //Person::func2(); 类外不可访问
29 }
30 int main()
31 {
32     test01();
33     system("pause");
34     return 0;
35 }

```

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

```
1  #include <iostream>
2  using namespace std;
3
4  //成员变量和成员函数分开存储
5  class Person {
6  public:
7      int m_A; //非静态成员变量属于类的对象
8      static int m_B; //静态成员变量不属于类的对象
9      void func() {}; //非静态成员函数不属于类的对象
10     static void func2() {}; //静态成员函数不属于类的对象
11 };
12 int Person::m_B = 100;
13 void test01() {
14     Person p;
15     //空对象占用内存空间为1
16     //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
17     cout << "sizeof(p) = " << sizeof(p) << endl;
18 }
19 void test02() {
20     Person p;
21     cout << "sizeof(p) = " << sizeof(p) << endl;
22 }
23 int main()
24 {
25     test01();
26     test02();
27     system("pause");
28     return 0;
29 }
```

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

```
1  #include <iostream>
2  using namespace std;
3
4  //this指针
5  class Person {
6  public:
7      Person(int age) {
8          //this指针指向被调用的成员函数所属的对象
9          this->m_Age = age;
10     }
11 }
```

```

11     Person &PersonAddAge(Person& p) {
12         this->m_Age += p.m_Age;
13         return *this;
14     }
15     int m_Age;
16 };
17 //1、解决名称冲突
18 void test01() {
19     Person p1(18);
20     cout << "p1.m_Age = " << p1.m_Age << endl;
21 }
22 //2、返回对象本身
23 void test02() {
24     Person p1(10);
25     Person p2(10);
26     //链式编程思想
27     p2.PersonAddAge(p1).PersonAddAge(p1).PersonAddAge(p1);
28     cout << "p2.m_Age = " << p2.m_Age << endl;
29 }
30
31 int main()
32 {
33     //test01();
34     test02();
35     system("pause");
36     return 0;
37 }

```

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

```

1  #include <iostream>
2  using namespace std;
3
4  //空指针调用成员函数
5  class Person {
6  public:
7      void showClassName() {
8          cout << "This is Person Class" << endl;
9      }
10
11     void showPersonAge() {
12         //报错原因是因为传入指针为空
13         if (this == NULL) { return; }
14         cout << "m_Age = " << this->m_Age << endl;
15     }
16     int m_Age;
17 };
18
19 void test01() {
20     Person* p = NULL;
21     p->showClassName();
22     //p->showPersonAge();
23 }

```

```

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

```
1  #include <iostream>
2  using namespace std;
3  #include <string>
4  //全局函数做友元
5  class Building {
6      friend void GoodGay(Building* building); //友元全局函数
7  public:
8      Building() {
9          m_SittingRoom = "客厅";
10         m_BedRoom = "卧室";
11     }
12     string m_SittingRoom;
13 private:
14     string m_BedRoom;
15 };
16 void GoodGay(Building* building) {
17     cout << "GoodGay() 正在访问" << building->m_SittingRoom << endl;
18     cout << "GoodGay() 正在访问" << building->m_BedRoom << endl;
19 }
20 void test01() {
21     Building building;
22     GoodGay(&building);
23 }
24 int main()
25 {
26     test01();
27     system("pause");
28     return 0;
29 }
```

4.4.2 类做友元

```
1  #include <iostream>
2  using namespace std;
3  #include <string>
4  //类做友元
5  class Building {
6      friend class GoodGay; //友元类
7  public:
8      Building();
9      string m_SittingRoom;
10 private:
11     string m_BedRoom;
```

```

12 };
13 class GoodGay {
14 public:
15     GoodGay();
16     void visit();
17     Building* building;
18 };
19 //类外实现成员函数
20 Building::Building() {
21     m_BedRoom = "卧室";
22     m_SittingRoom = "客厅";
23 }
24 GoodGay::GoodGay() {
25     building = new Building;
26 }
27 void GoodGay::visit() {
28     cout << "GoodGay() 正在访问" << building->m_SittingRoom << endl;
29     cout << "GoodGay() 正在访问" << building->m_BedRoom << endl;
30 }
31
32 void test01() {
33     GoodGay gg;
34     gg.visit();
35 }
36 int main()
37 {
38     test01();
39     system("pause");
40     return 0;
41 }

```

4.4.3 成员函数做友元

```

1  #include <iostream>
2  using namespace std;
3  #include <string>
4  //成员函数做友元
5  class Building;
6  class GoodGay {
7  public:
8      GoodGay();
9      void visit();//友元函数
10     void visit2();
11     Building* building;
12 };
13 class Building {
14     friend void GoodGay::visit();//友元函数
15 public:
16     Building();
17     string m_SittingRoom;
18 private:
19     string m_BedRoom;
20 };
21 Building::Building() {
22     m_BedRoom = "卧室";
23     m_SittingRoom = "客厅";
24 }

```

```

25 GoodGay::GoodGay() {
26     building = new Building;
27 }
28 void GoodGay::visit() {
29     cout << "GoodGay::visit() 正在访问" << building->m_SittingRoom << endl;
30     cout << "GoodGay::visit() 正在访问" << building->m_BedRoom << endl;
31 }
32 void GoodGay::visit2() {
33     cout << "GoodGay::visit2() 正在访问" << building->m_SittingRoom << endl;
34 }
35 void test01() {
36     GoodGay gg;
37     gg.visit();
38     gg.visit2();
39 }
40 int main()
41 {
42     test01();
43     system("pause");
44     return 0;
45 }

```

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

```

1  #include <iostream>
2  using namespace std;
3  #include <string>
4  //加号运算符重载
5  class Person {
6  public:
7      //成员函数重载+
8      /*Person operator+(Person&p) {
9          Person temp;
10         temp.m_A = this->m_A + p.m_A;
11         temp.m_B = this->m_B + p.m_B;
12         return temp;
13     }*/
14     int m_A;
15     int m_B;
16 };
17 //全局函数重载+
18 Person operator+(Person& p1, Person& p2) {
19     Person temp;
20     temp.m_A = p1.m_A + p2.m_A;
21     temp.m_B = p1.m_B + p2.m_B;
22     return temp;
23 }
24 //函数重载的版本
25 Person operator+(Person& p1, int num) {
26     Person temp;
27     temp.m_A = p1.m_A + num;

```

```

28     temp.m_B = p1.m_B + num;
29     return temp;
30 }
31 void test01() {
32     Person p1;
33     p1.m_A = 10;
34     p1.m_B = 20;
35     Person p2;
36     p2.m_A = 10;
37     p2.m_B = 20;
38
39     Person p3 = p1 + p2;
40     //运算符重载也可以发生函数重载
41     Person p4 = p1 + 10;
42 }
43 int main()
44 {
45     test01();
46     system("pause");
47     return 0;
48 }

```

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

```

1  #include <iostream>
2  using namespace std;
3  #include <string>
4  //左移运算符重载
5  class Person {
6      friend ostream& operator<<(ostream& cout, Person& p);
7  public:
8      Person(int a, int b){
9          m_A = a;
10         m_B = b;
11     }
12 private:
13     int m_A;
14     int m_B;
15 };
16 //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
17 ostream& operator<<(ostream& cout, Person& p) { //本质 operator << (cout, p) --
18     >cout<<p<<cout << "p.m_A = " << p.m_A << endl;
19     cout << "p.m_A = " << p.m_A << endl;
20     cout << "p.m_B = " << p.m_B << endl;
21     return cout; //实现链式编程
22 }
23 void test01() {
24     Person p(10, 10);
25     cout << p << endl; //本质 operator<<(cout, p);
26 }
27 int main()
28 {

```



```

28     test01();
29     system("pause");
30     return 0;
31 }

```

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

```

1  #include <iostream>
2  using namespace std;
3  //递增运算符重载
4  class MyInteger {
5      friend ostream& operator<<(ostream& cout, MyInteger myint);
6  public:
7      MyInteger() {
8          m_Num = 0;
9      }
10     //前置递增
11     MyInteger& operator++() {
12         //先递增
13         m_Num++;
14         //再将自身返回,返回引用，一直对一个数操作
15         return *this;
16     }
17     //后置递增,int 代表占位参数，用于区分前置和后置递增
18     MyInteger operator++(int) {
19         //先记录当时结果
20         MyInteger temp = *this;
21         //再递增
22         m_Num++;
23         //再将记录的结果返回，返回值（不可返回局部对象的引用）
24         return temp;
25     }
26 private:
27     int m_Num;
28 };
29 //重载<<
30 ostream& operator<<(ostream& cout, MyInteger myint) {
31     cout << myint.m_Num;
32     return cout;
33 }
34 void test01() {
35     MyInteger myint;
36     cout << ++(++myint) << endl;
37     cout << myint << endl;
38 }
39 void test02() {
40     MyInteger myint;
41     cout << myint++ << endl;
42     cout << myint << endl;
43 }
44 int main() {
45     test01();
46     test02();
47     system("pause");
48     return 0;

```

- 重载递减运算符

```

1  #include <iostream>
2  using namespace std;
3  //递减运算符重载
4  class MyInter {
5      friend ostream& operator<<(ostream& cout, MyInter myint);
6  private:
7      int m_Num;
8  public:
9      MyInter() {
10         m_Num = 0;
11     }
12     MyInter& operator--() {
13         m_Num--;
14         return *this; //返回自身
15     }
16     MyInter operator--(int) {
17         MyInter temp = *this; //先保存当前的值
18         m_Num--;
19         return temp;
20     }
21 };
22 ostream& operator<<(ostream&cout,MyInter myint) {
23     cout << myint.m_Num;
24     return cout;
25 }
26 void test01() {
27     MyInter myint;
28     cout << --(--myint) << endl;
29     cout << (myint--)-- << endl;
30 }
31 int main() {
32     test01();
33     system("pause");
34     return 0;
35 }

```

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）

```

1  #include <iostream>
2  using namespace std;
3  //赋值运算符重载
4  class Person {
5  public:
6      Person(int age) {
7          m_Age = new int(age);

```

```

8     }
9     Person& operator=(Person& p) {
10         //m_Age = p.m_Age; 编译器提供
11         //先判断堆区是否干净
12         if (m_Age != NULL) {
13             delete m_Age;
14             m_Age = NULL;
15         }
16         this->m_Age = new int(*p.m_Age); //深拷贝
17         return *this; //返回对象本身，链式编程
18     }
19     ~Person() {
20         if (m_Age != NULL) {
21             delete m_Age;
22             m_Age = NULL;
23         }
24     }
25     int* m_Age;
26 };
27 void test01() {
28     Person p1(18);
29     Person p2(19);
30     Person p3(30);
31     p3 = p2 = p1;
32     cout << "*p1.m_Age = " << *p1.m_Age << endl;
33     cout << "*p2.m_Age = " << *p2.m_Age << endl;
34     cout << "*p3.m_Age = " << *p3.m_Age << endl;
35 }
36 int main() {
37     test01();
38     system("pause");
39     return 0;
40 }

```

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

```

1  #include <iostream>
2  using namespace std;
3  #include <string>
4  //关系运算符重载
5  class Person {
6  public:
7      Person(string name, int age) {
8          m_Age = age;
9          m_Name = name;
10     }
11     bool operator==(Person& p) {
12         if (p.m_Name == this->m_Name && p.m_Age == this->m_Age) {
13             return true;
14         }
15         return false;
16     }
17     bool operator!=(Person& p) {
18         if (p.m_Name == this->m_Name && p.m_Age == this->m_Age) {
19             return false;

```

```

20     }
21     return true;
22 }
23 string m_Name;
24 int m_Age;
25 };
26 void test01() {
27     Person p1("Tom", 18);
28     Person p2("Tom", 18);
29     if (p1 != p2) {
30         cout << "p1 != p2" << endl;
31     }
32     else {
33         cout << "p1 == p2" << endl;
34     }
35 }
36 int main() {
37     test01();
38     system("pause");
39     return 0;
40 }

```

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：

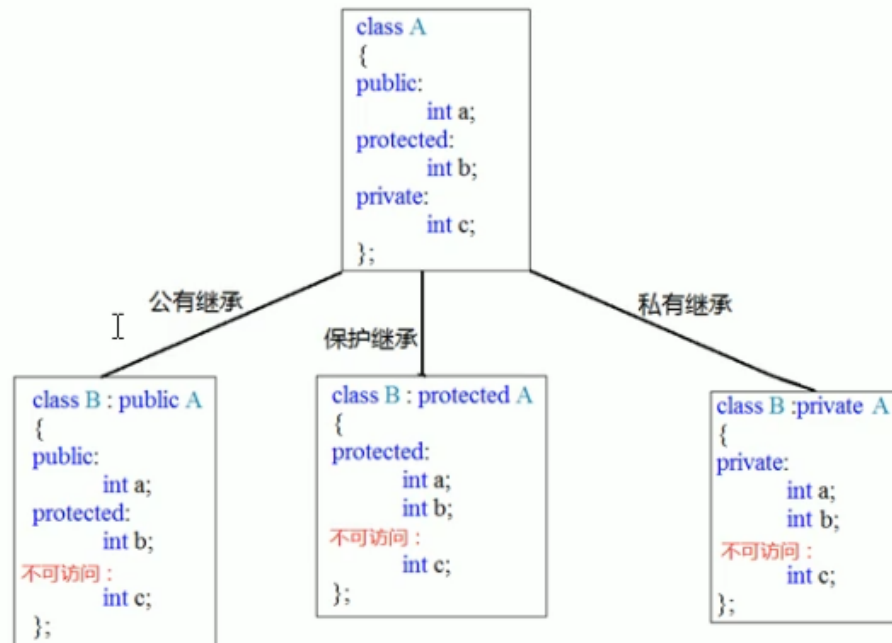
```

1 class 子类: 继承方式 父类
2     //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

```

1  #include <iostream>
2  using namespace std;
3  #include <string>
4  //继承中的对象模型
5  class Base {
6  public:
7      int m_A;
8  protected:
9      int m_B;
10 private:
11     int m_C;
12 };
13 class Son :public Base {
14 public:
15     int m_D;
16 };
17 void test01() {
18     //父类中所有非静态成员属性都会被子类继承下去
19     //父类中的私有属性被编译器隐藏，因此无法访问
20     cout << "sizeof(Son) = " << sizeof(Son) << endl;
21 }
22 int main() {
23     test01();
24     system("pause");
25     return 0;
26 }

```

- 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

```
1  #include <iostream>
2  using namespace std;
3  #include <string>
4  //继承中的对象模型
5  class Base {
6  public:
7      Base() {
8          cout << "Base构造函数调用" << endl;
9      }
10     ~Base() {
11         cout << "Base析构函数调用" << endl;
12     }
13 };
14 class Son :public Base {
15 public:
16     Son() {
17         cout << "Son构造函数调用" << endl;
18     }
19     ~Son() {
20         cout << "Son析构函数调用" << endl;
21     }
22 };
23 void test01() {
24     //创建子类对象时，先构造父类再构造子类
25     //析构顺序与构造顺序相反
26     Son s1;
27 }
28 int main() {
29     test01();
30     system("pause");
31     return 0;
32 }
```

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

```
1  #include <iostream>
2  using namespace std;
3  #include <string>
4  //继承中同名成员处理
5  class Base {
6  public:
7      Base() {
8          m_A = 100;
9      }
10     void func() {
11         cout << "Base::func()调用" << endl;
12     }
13     void func(int a) {
```

```

14     cout << "Base::func(int a)调用" << endl;
15 }
16 int m_A;
17 };
18 class Son :public Base {
19 public:
20     Son() {
21         m_A = 200;
22     }
23     void func() {
24         cout << "Son::func()调用" << endl;
25     }
26     int m_A;
27 };
28 void test01() {
29     Son s;
30     //子类访问子类, 直接访问
31     cout << "Son::m_A = " << s.m_A << endl;
32     s.func();
33     //子类访问父类同名成员, 加作用域
34     cout << "Base::m_A = " << s.Base::m_A << endl;
35     s.Base::func();
36
37     //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
38     //访问加作用域
39     s.Base::func(100);
40 }
41 int main() {
42     test01();
43     system("pause");
44     return 0;
45 }

```

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

```

1  #include <iostream>
2  using namespace std;
3  #include <string>
4  //继承同名静态成员处理
5  class Base {
6  public:
7      static int m_A;
8      static void func() {
9          cout << "Base::func()调用" << endl;
10     }
11     static void func(int a) {
12         cout << "Base::func(int a)调用" << endl;
13     }
14 };
15 int Base::m_A = 100;
16 class Son :public Base {
17 public:

```

```

18     static int m_A;
19     static void func() {
20         cout << "Son::func()调用" << endl;
21     }
22
23 };
24 int Son::m_A = 200;
25 void test01() {
26     //通过对象访问
27     Son s;
28     cout << "通过对象访问: " << endl;
29     cout << "Son::m_A = " << s.m_A << endl;
30     cout << "Base::m_A = " << s.Base::m_A << endl;
31     s.func();
32     s.Base::func();
33     //通过类名
34     cout << "通过类名访问: " << endl;
35     cout << "Son::m_A = " << Son::m_A << endl;
36     //cout << "Base::m_A = " << Base::m_A << endl;
37     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
38     Son::func();
39     Son::Base::func();
40     Son::Base::func(100);
41 }
42 int main() {
43     test01();
44     system("pause");
45     return 0;
46 }

```

4.6.7 多继承语法

C++允许一个类继承多个类

语法:

```
1 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)

```

1 #include <iostream>
2 using namespace std;
3 #include <string>
4 //菱形继承
5

```



```

6 //动物类-->虚基类
7 class Animal{
8 public:
9     int m_Age;
10 };
11 //利用虚继承解决菱形继承的问题
12 //羊类
13 class Sheep:virtual public Animal{};
14 //驼类
15 class Camel:virtual public Animal{};
16 //羊驼类
17 class Alpaca :public Sheep, public Camel {};
18 void test01() {
19     Alpaca a1;
20     a1.Sheep::m_Age = 18;
21     a1.Camel::m_Age = 28;
22     cout << "a1.Sheep::m_Age = " << a1.Sheep::m_Age << endl;
23     cout << "a1.Camel::m_Age = " << a1.Camel::m_Age << endl;
24     cout << "a1.m_Age = " << a1.m_Age << endl;
25 }
26 int main() {
27     test01();
28     system("pause");
29     return 0;
30 }

```

```

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

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

Alpaca::$vtable@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

```
1  #include <iostream>
2  using namespace std;
3  #include <string>
4  //多态
5  class Animal {
6  public:
7      //虚函数,函数地址晚绑定
8      virtual void Speak() {
9          cout << "动物在说话" << endl;
10     }
11 };
12 class Cat :public Animal {
13 public:
14     void Speak() {
15         cout << "小猫在说话" << endl;
16     }
17 };
18 class Dog :public Animal {
19 public:
20     void Speak() {
21         cout << "小狗在说话" << endl;
22     }
23 };
24 //动态多态满足条件
25 //1、继承关系
26 //2、子类重写父类的虚函数
27 //动态多态的使用
28 //父类的指针或者引用指向子类对象
29 void doSpeak(Animal& animal) {
30     animal.Speak();
31 }
32 void test01() {
33     Cat cat;
34     doSpeak(cat); //Animal& animal = cat;
35     Dog dog;
36     doSpeak(dog);
37 }
38 int main() {
39     test01();
40     system("pause");
41     return 0;
42 }
```

```

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

```
1  #include <iostream>
2  using namespace std;
3
4  //多态写法
5  //开发中：开闭原则（对扩展开放，对修改封闭）
6  //计算器基类
7  class AbstractCalculator {
8  public:
9      virtual int getResult() {
10         return 0;
11     }
12
13     int m_Num1;
14     int m_Num2;
15 };
16
17 //加法计算器类
18 class AddCalculator :public AbstractCalculator {
19 public:
20     int getResult() {
21         return m_Num1 + m_Num2;
22     }
23 };
24 //减法计算器类
25 class SubCalculator :public AbstractCalculator {
26 public:
27     int getResult() {
28         return m_Num1 - m_Num2;
29     }
30 };
31 //乘法计算器类
32 class MulCalculator :public AbstractCalculator {
33 public:
34     int getResult() {
35         return m_Num1 * m_Num2;
36     }
37 };
38
39 void test01() {
40     //多态使用条件：父类的指针或者引用指向子类对象
41     //加法
42     AbstractCalculator* abs = new AddCalculator;
43     abs->m_Num1 = 100;
44     abs->m_Num2 = 10;
45     cout << abs->m_Num1 << "+" << abs->m_Num2 << "=" << abs->getResult() <<
endl;
46     delete abs;
47     //减法
48     abs = new SubCalculator;
```

```

49     abs->m_Num1 = 100;
50     abs->m_Num2 = 10;
51     cout << abs->m_Num1 << "-" << abs->m_Num2 << "=" << abs->getResult() <<
endl;
52     delete abs;
53     //乘法
54     abs = new MulCalculator;
55     abs->m_Num1 = 100;
56     abs->m_Num2 = 10;
57     cout << abs->m_Num1 << "*" << abs->m_Num2 << "=" << abs->getResult() <<
endl;
58     delete abs;
59     abs = NULL;
60 }
61 int main() {
62     test01();
63
64     system("pause");
65     return 0;
66 }

```

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：

```

1 | 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

```

1 | #include <iostream>
2 | using namespace std;
3 |
4 | //纯虚函数和抽象类
5 | class Base {
6 | public:
7 |     virtual void func() = 0; //纯虚函数
8 | };
9 | class Son :public Base {
10 | public:
11 |     //子类必须重写父类中的纯虚函数，否则也属于抽象类
12 |     void func() {
13 |         cout << "func()调用" << endl;
14 |     }
15 | };
16 | void test01() {
17 |     //Base b;    抽象类无法实例化对象
18 |     //new Base; 抽象类无法实例化对象
19 |     Base* base = new Son;
20 |     base->func();
21 |     delete base;

```

```

22 }
23 int main() {
24     test01();
25
26     system("pause");
27     return 0;
28 }

```

4.7.4 多态案例二——制作饮品

```

1  #include <iostream>
2  using namespace std;
3
4  //制作饮品
5  class AbstractDrinking {
6  public:
7      virtual void Boil() = 0; //煮水
8      virtual void Brew() = 0; //冲泡
9      virtual void PourInCup() = 0; //倒入杯中
10     virtual void PutSomething() = 0; //加入辅料
11     //制作饮品
12     void MakeDrink() {
13         Boil();
14         Brew();
15         PourInCup();
16         PutSomething();
17     }
18 };
19 class Coffee : public AbstractDrinking {
20     void Boil() {
21         cout << "煮水" << endl;
22     }
23     void Brew() {
24         cout << "冲泡咖啡" << endl;
25     }
26     void PourInCup() {
27         cout << "倒入杯中" << endl;
28     }
29     void PutSomething() {
30         cout << "加入牛奶和糖" << endl;
31     }
32 };
33 class LenmonTea : public AbstractDrinking {
34     void Boil() {
35         cout << "煮水" << endl;
36     }
37     void Brew() {
38         cout << "冲泡茶叶" << endl;
39     }
40     void PourInCup() {
41         cout << "倒入杯中" << endl;
42     }
43     void PutSomething() {
44         cout << "加入柠檬" << endl;
45     }
46 };
47 void doWork(AbstractDrinking* abs) {

```

```

48     abs->MakeDrink();
49     delete abs;
50 }
51 void test01() {
52     cout << "制作咖啡" << endl;
53     dowork(new Coffee);
54     cout << endl << "制作柠檬茶" << endl;
55     dowork(new LenmonTea);
56 }
57 int main() {
58     test01();
59
60     system("pause");
61     return 0;
62 }

```

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

```

1 //虚析构语法：
2 virtual ~类名(){}
3 //纯虚析构语法
4 virtual ~类名() = 0;
5 类名::~~类名(){}

```

```

1 #include <iostream>
2 using namespace std;
3 #include <string>
4 //虚析构和纯虚析构
5 class Animal {
6 public:
7     Animal() {
8         cout << "Animal构造函数调用" << endl;
9     }
10    virtual void speak() = 0;
11
12    //纯虚析构---需要声明也需要实现
13    //有了纯虚析构，这个类也属于抽象类
14    virtual ~Animal() = 0;
15    //虚析构——解决父类指针释放子类对象不干净的问题
16    /*virtual ~Animal() {
17        cout << "Animal析构函数调用" << endl;
18    }*/
19 };
20 Animal::~~Animal() {
21     cout << "Animal纯虚析构函数调用" << endl;

```

```

22 }
23 class Cat :public Animal {
24 public:
25     Cat(string name) {
26         cout << "Cat构造函数调用" << endl;
27         m_Name = new string(name);
28     }
29     virtual void speak() {
30         cout << *m_Name << "小猫在说话" << endl;
31     }
32     ~Cat() {
33         cout << "Cat析构函数调用" << endl;
34         if (m_Name != NULL) {
35             delete m_Name;
36             m_Name = NULL;
37         }
38     }
39     string* m_Name;
40 };
41
42 void test01() {
43     Animal* animal = new Cat("Tom");
44     animal->speak();
45     delete animal;
46 }
47 int main() {
48     test01();
49
50     system("pause");
51     return 0;
52 }

```

4.7.6 多态案例三——组装电脑

```

1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  //抽象的CPU类
7  class CPU {
8  public:
9      //抽象的计算函数
10     virtual void calculate() = 0;
11 };
12 //抽象的显卡类
13 class VideoCard {
14 public:
15     //抽象的显示函数
16     virtual void display() = 0;
17 };
18 //抽象的内存条类
19 class Memory {
20 public:
21     //抽象的存储函数
22     virtual void storage() = 0;
23 };

```



```

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:

```

```

82     void calculate() {
83         cout << "Lenovo的CPU开始计算了" << endl;
84     }
85 };
86 class LenovoVedioCard :public VideoCard {
87 public:
88     void display() {
89         cout << "Lenovo的显卡开始显示了" << endl;
90     }
91 };
92 class LenovoMemory :public Memory {
93 public:
94     void storage() {
95         cout << "Lenovo的内存条开始存储了" << endl;
96     }
97 };
98 void test01() {
99     //第一台电脑
100    CPU* intelCpu = new IntelCpu;
101    VideoCard* intelVedioCard = new IntelVedioCard;
102    Memory* intelMemory = new IntelMemory;
103
104    //创建第一台电脑
105    Computer* computer1 = new Computer(intelCpu, intelVedioCard,
intelMemory);
106    computer1->work();
107    delete computer1;
108
109    //创建第二台电脑
110    Computer* computer2 = new Computer(new LenovoCpu, new LenovoVedioCard,
new LenovoMemory);
111    computer2->work();
112    delete computer2;
113 }
114 int main() {
115     test01();
116
117     system("pause");
118     return 0;
119 }

```

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：

- ofstream 写操作
- ifstream 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

```
1 #include <fstream>
2 ofstream ofs;
3 ofs.open("文件路径", 打开方式);
4 ofs << "写入的数据";
5 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

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 //写文件
6 void test01() {
7     ofstream ofs;//创建文件流
8     ofs.open("test.txt", ios::out);//打开文件
9     ofs << "姓名：张三" << endl
10         << "年龄：18" << endl
11         << "性别：男" << endl;//写入数据
12     ofs.close();//关闭文件
13 }
14 int main()
15 {
16     test01();
17     system("pause");
18     return 0;
19 }
```

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开成功
- 读数据（四种方式读取）
- 关闭文件

```
1  #include <iostream>
2  #include <string>
3  #include <fstream>
4  using namespace std;
5
6  //读文件
7  void test01() {
8      ifstream ifs; //创建流对象
9      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
10     if (!ifs.is_open()) {
11         cout << "文件打开失败" << endl;
12         return;
13     }
14     //读文件
15
16     //第一种
17     char buffer[1024] = { 0 };
18     while (ifs >> buffer) {
19         cout << buffer << endl;
20     }
21
22     //第二种
23     char buffer[1024] = { 0 };
24     while (ifs.getline(buffer, sizeof(buffer))) {
25         cout << buffer << endl;
26     }
27
28     //第三种
29     string buffer;
30     while (getline(ifs, buffer)) {
31         cout << buffer << endl;
32     }
33
34     //第四种
35     char c;
36     while ((c = ifs.get()) != EOF) {
37         cout << c;
38     }
39     ifs.close();
40 }
41
42 int main()
43 {
44     test01();
45     system("pause");
46     return 0;
47 }
```

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);

```
1  #include <iostream>
2  #include <string>
3  #include <fstream>
4  using namespace std;
5
6  //二进制读文件
7  class Person {
8  public:
9      char m_Name[64];
10     int m_Age;
11 };
12 void test01() {
13     ofstream ofs("person.txt", ios::out | ios::binary); //创建流对象
14     //ofs.open("person.txt", ios::out | ios::binary);
15     Person p = { "Tom", 18 };
16     ofs.write((const char*)&p, sizeof(Person));
17     ofs.close();
18 }
19
20 int main()
21 {
22     test01();
23     system("pause");
24     return 0;
25 }
```

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);

```
1  #include <iostream>
2  #include <string>
3  #include <fstream>
4  using namespace std;
5
6  //二进制读文件
7  class Person {
8  public:
9      char m_Name[64];
10     int m_Age;
11 };
12 void test01() {
13     ifstream ifs;
14     ifs.open("person.txt", ios::in | ios::binary);
15     if (!ifs.is_open()) {
16         cout << "文件打开失败" << endl;
17         return;
18     }
19 }
```

```
18     }
19     Person p;
20     ifs.read((char*)&p, sizeof(Person));
21     cout << "姓名: " << p.m_Name << endl;
22     cout << "年龄: " << p.m_Age << endl;
23     ifs.close();
24 }
25
26 int main()
27 {
28     test01();
29     system("pause");
30     return 0;
31 }
```