一. 类和对象的基本概念
1.1 c和c++中struct的区别
1.2 C语言中表示事物的方法存在的问题
1.3 c++中对事物的封装-类
1.4 类中的成员权限
1.5 尽量设置成员变量为私有权限
1.6 课堂练习
二. 面向对象程序设计案例
2.1 立方体案例
2.2 点和圆的关系
三. 构造和析构
3.1 构造和析构的概念
3.2 构造函数和析构函数
3.3 构造函数的分类
3.4 拷贝构造函数的调用时机
3.5 c++默认增加的函数
3.6 构造函数的浅拷贝和深拷贝
3.6.1 浅拷贝
3.7多个对象构造和析构
3.7.1 初始化列表
3.7.1 例如 10列表

3.8 explicit
3.9 动态对象创建
3.9.1 malloc和free动态申请对象和释放对象
3.9.1 3.9.2 c++中动态申请对象和释放对象
3.9.3 用于数组的new和delete
3.9.4 delete void *问题
3.9.5 使用new和delete采用相同形式
3.10 静态成员
3.10.1 静态成员变量
3.10.2 静态成员函数
3.10.3 const修饰的静态成员变量
3.10.4 单例模式
四. 类对象成员的初探
1.成员变量和函数的存储
2 this指针的工作原理
3this指针的应用
4 const修饰的成员函数
五:友元
1 友元的语法
1.1全局函数成为类的友元
1.2 类成为另一个类的友元,类的成员函数成为另一个类的友元
六: 运算符重载
1运算符重载的基本概念
2 重载加号运算符

3 重载左移运算符和算符重载碰上友元函数
4可以重载的运算符
5 重载自加自减运算符
6 智能指针
7 <u>重</u> 载=号运算符
8 重载等于和不等于号
9 函数对象
10 不要重载&&和
11 重载运算符建议
12 封装string类
13 优先级
七.继承
1继承的概念
1.1 为什么需要继承
1.2 继承的概念
1.3 派生类的定方义法
2 派生类访问权限控制
3 继承中的析构和构造
3.1 继承中的对象模型
3.2 对象构造和析构的调用原则
4.继承中同名成员的处理问题
5 非自动继承的函数
6 继承中的静态成员特性
7 多继承

7.1 多继承的概念
7.2菱形继承和虚继承
7.3虚继承的实现原理
八.多态
1 多态的概念
2 多态实现计算器的案例
3 c++如何实现动态绑定
4 纯虚函数和抽象类
5 纯虚函数和多继承
6 虚析构
7 纯虚析构
8 重载 重定义 重写

一. 类和对象的基本概念

1.1 c和c++中struct的区别

- c语言中结构体中不能存放函数,也就是数据(属性)和行为(方法)是分离的
- c++中结构体中是可以存放函数的,也就是数据(属性)和行为 (方法)是封装在一起的

```
#define _CRT_SECURE_NO_WARNINGS

#include <iostream>
#include <string.h>

using namespace std;

//c语言中 不能放函数

struct _stu

{

int a;

int b[5];

int b[5];
```

```
12 //c++中 可以放函数
13 struct _stu1
14 {
15 int a;
16 int b[5];
  void print_stu()
18
   cout << a << endl;</pre>
19
20
   }
21
22
  };
23
24 struct student
25 {
  //学生的属性和数据
26
  int age;
27
  int id;
28
29 char name[256];
   //操作属性的叫做 方法或行为-函数
  void print()
31
32
   {
33
   cout << age << id << name << endl;</pre>
34
35
36
37 };
38 void test01()
39 {
  student obj;
40
41 obj.age = 10;
42 obj.id = 20;
43 strcpy(obj.name,"lucy");
44 obj.print();
45
46 }
47 int main()
48 {
  test01();
49
50
  return 0;
51 }
```

1.2 C语言中表示事物的方法存在的问题

c语言中表示事物时,将属性和行为分离,有可能行为调用出错

```
1 #include <stdio.h>
2 //表示人
3 struct Person
4 {
5 int age;
6 char name[128];
8 void Person_eat(struct Person *p)
9 {
10 printf("%s 在吃饭\n",p->name);
11 }
12
13 //表示dog
14 struct Dog
15 {
16 int age;
17 char name[128];
18 };
19 void Dog_eat(struct Dog *p)
20 {
21 printf("%s 在吃粑粑\n", p->name);
22 }
24
25 void test01()
26 {
27 struct Person p1;
   p1.age = 20;
   strcpy(p1.name, "bob");
29
    Person_eat(&p1);
30
31
   struct Dog d1;
32
   d1.age = 7;
33
   strcpy(d1.name,"旺财");
35
    Dog_eat(&d1);
36
```

```
37 Dog_eat(&p1);//人 调用了狗的行为
38
39
40 }
41 int main()
42 {
43 test01();
44 return 0;
45 }
```

1.3 c++中对事物的封装-类

- c++将事物的属性和行为封装在一起
- 类和结构体的一个区别在于,类对成员可以进行访问的权限控制,结构体不可以
- 类 = 类型 (事物的行为和属性) 类实例化出来的变量叫对象
- 类中的函数 可以访问类里面的成员

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 using namespace std;
5 //c++中对事物的封装 将属性和行为封装在一起
6 //类 将事物抽象成属性和行为,并且封装在一起
7 //结构体中所有成员默认都是公有的 类中的所有成员默认是私有的,也可以修改成员的访问
权限
8 // struct Person
9 class Person
10 {
11 public://公有的
  //类中的所有成员 访问的权限都是私有的 private
13 //属性
14 int age;
15 char name[128];
16 //行为
17 void Person_eat()
18 {
  printf("%s 吃饭\n",name);
19
  }
20
21 };
```

```
22
23 struct Dog
24 {
  //属性
25
26
   int age;
   char name[128];
28
   //行为
  void Dog_eat()
29
30
   printf("%s 吃粑粑\n", name);
31
   }
32
33
  };
34
  void test01()
36 {
   //通过类 实例化出一个变量 这个变量叫对象
37
   Person p1;
38
   p1.age = 10;
39
  strcpy(p1.name, "lucy");
40
   p1.Person_eat();
41
42
  Dog d1;
43
44 d1.age == 6;
45 strcpy(d1.name,"旺财");
   d1.Dog_eat();
46
47
48
  }
49
50
51
  int main()
52
53
54
   test01();
55
56
57 }
```

1.4 类中的成员权限

访问属性	属性	对象内部	对象外部
public	公有	可访问	可访问
protected	保护	可访问	不可访问
private +	私有	可访问	不可访问

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 using namespace std;
5 class person
7 public: //公有的 类内类外都可访问
  int mTall; //多高,可以让外人知道
9 protected: // 保护的 类外不可访问 类内是可以访问的 子类可以访问
   int mMoney; // 有多少钱,只能儿子孙子知道
  private: //私有的 类外不可访问 类内是可以访问的 子类不可访问
   int mAge; //年龄, 不想让外人知道
12
13
  void show()
14
15 {
16 cout << mTall << " ";</pre>
17 cout << mMoney << " ";</pre>
  cout << mAge << " ";
18
   }
19
20
21 };
22 void test01()
23 {
24
  person p;
   p.mTall = 180;
26
27
28
  }
29
30 int main()
31 {
32
   return 0;
34 }
```

1.5 尽量设置成员变量为私有权限

设置成员变量为私有,优点:

- 对变量的设置时的控制
- 可以给变量设置只读权限
- 可以给变量设置只写权限
- 可以给变量设置可读可写权限

```
1 class AccessLevels{
2 public:
3 //对只读属性进行只读访问
4 int getReadOnly(){ return readOnly; }
5 //对读写属性进行读写访问
6 void setReadWrite(int val){ readWrite = val; }
7 int getReadWrite(){ return readWrite; }
8 //对只写属性进行只写访问
9 void setWriteOnly(int val){ writeOnly = val; }
10 private:
11 int readOnly; //对外只读访问
12 int noAccess; //外部不可访问
13 int readWrite; //读写访问
14 int writeOnly; //只写访问
15 };
```

1.6 课堂练习

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>

using namespace std;

class Person

{
public:

void person_init(int age, char *name)

{

if (age >= 0 && age <= 100)

m_age = age;

strcpy(m_name, name);

}

void show_person()</pre>
```

```
15
    cout << m_name << " " << m_age << endl;</pre>
17
    int get_age()
18
19
    return m_age;
20
21
    void set_age(int age)
22
23
    if (age >= 0 && age <= 100)</pre>
24
25
    m_age = age;
26
27
28
    char *get_name()
29
30
    return m_name;
31
32
33
    void set_name(char *name)
34
    strcpy(m_name,name);
    }
36
37
38 private:
   int m_age;
39
    char m_name[128];
40
41
42
   };
43
44 void test01()
45 {
    Person p1;
46
   p1.person_init(20,"lucy");
47
   p1.show_person();
48
49
   p1.set_age(30);
   p1.set_name("bob");
51
    p1.show_person();
53 int main()
54 {
```

```
55  test01();
56  return 0;
57 }
```

二. 面向对象程序设计案例

2.1 立方体案例

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 using namespace std;
5 class Cube
6 {
7 public:
8 void set_L(int 1)
10 \quad L = 1;
11 }
12
  void set_W(int w)
13 {
14 \quad W = W;
15 }
  void set_H(int h)
16
17 {
   H = h;
19
   int get_L()
21
22
   return L;
23
   int get_W()
24
25
   return W;
26
27
   int get_H()
28
29
   {
30
   return H;
31
   //求立方体的体积
32
33 int get_cube_V()
```

```
34
    return L*W*H;
36
    //求立方体面积
37
    int get_cube_S()
38
    return 2 * W*L + 2*W*H + 2 * L*H;
40
41
    }
    //判断两个立方体是否相等
42
    bool compare_cube(Cube &c1)
43
44
    return c1.get_L() == L && c1.get_W() == W && c1.get_H() == H;
45
46
47
48
   private:
49
    int L;
50
    int W;
51
    int H;
   };
52
53
   bool comapre_cube(Cube &c1, Cube &c2)
54
   {
55
56
    return c1.get_L() == c2.get_L()&& c1.get_W() == c2.get_W() &&
c1.get_H() == c2.get_H();
58 }
59
  void test01()
61 {
62
    Cube c1;
    c1.set_L(10);
63
    c1.set_W(20);
64
    c1.set_H(30);
65
    cout << c1.get_cube_S() << endl;</pre>
66
    cout << c1.get_cube_V() << endl;</pre>
67
    Cube c2;
69
    c2.set_L(20);
70
71
    c2.set_W(20);
    c2.set_H(30);
72
73
```

```
if (c1.compare_cube(c2))
   {
75
76
77
   cout << "立方体相等" << endl;
78
79
   else
   {
80
   cout << "立方体不相等" << endl;
81
    }
82
83
   if (comapre_cube(c1, c2))
84
85
   cout << "立方体相等" << endl;
86
87
   }
   else
88
89
   cout << "立方体不相等" << endl;
90
   }
91
92
93
94
95 }
96 int main()
97 {
  test01();
98
   return 0;
99
100 }
```

2.2 点和圆的关系

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
using namespace std;

//定义点类
class Point {
public:
void setX(int x) { mX = x; }

void setY(int y) { mY = y; }

int getX() { return mX; }
```

```
int getY() { return mY; }
13
  private:
   int mX;
14
   int mY;
16 };
17
  //圆类
18
19 class Circle {
20 public:
    void setP(int x, int y) {
21
    mP.setX(x);
    mP.setY(y);
23
    }
24
    void setR(int r) { mR = r; }
25
26
    Point& getP() { return mP; }
27
    int getR() { return mR; }
    //判断点和圆的关系
28
    void IsPointInCircle(Point& point) {
    int distance = (point.getX() - mP.getX()) * (point.getX() - mP.getX())
+ (point.getY() - mP.getY()) * (point.getY() - mP.getY());
    int radius = mR * mR;
    if (distance < radius) {</pre>
32
    cout << "Point(" << point.getX() << "," << point.getY() << ")在圆内!" <<
end1;
34
    else if (distance > radius) {
36 cout << "Point(" << point.getX() << "," << point.getY() << ")在圆外!" <<
end1;
37
    }
   else {
38
   cout << "Point(" << point.getX() << "," << point.getY() << ")在圆上!" <<
end1;
40
41
42 private:
   Point mP; //圆心
   int mR; //半径
44
45 };
46
47 void test() {
  //实例化圆对象
```

```
49 Circle circle;
50 circle.setP(20, 20);
51 circle.setR(5);
52 //实例化点对象
53 Point point;
54 point.setX(25);
55 point.setY(20);
56
57 circle.IsPointInCircle(point);
58 }
59 int main()
60 {
61 test();
62 return 0;
63 }
```

三. 构造和析构

3.1 构造和析构的概念

- 创建对象时,对对象进行初始化的工作,就是构造
- 销毁对象时,对对象进行清理工作,就是析构
- 一般需要人为提供,如果不提供,那么编译器也会给提供,只是编译器提供的构造和析构函数不会做任何操作
- 创建对象时和释放对象时,构造函数和析构函数自动会调用,不需要人为调用

3.2 构造函数和析构函数

构造函数:

- 没有返回值
- 函数名和类名一致
- 有参数,参数可以有多个
- 可以发送函数的重载
- 创建对象时,会自定调用

析构函数:

- 没有返回值
- 函数名: 类名前面加上~
- 没有参数
- 不能发送函数的重载
- 销毁对象之前,回被自动调用

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <string>
5 using namespace std;
7 class Person
8 {
9 public:
   //构造函数 1 函数名和类名一致 没有返回值 不能写void 可以有参数 可以发生函数重
10
载
   Person(int age, string name)
11
13 cout << "person的构造函数" << endl;
14 m_age = age;
15
  m_name = name;
16
  //析构函数 函数名: 类名前面加上~ 没有返回值 不可以有参数 不能发生函数重载
17
  ~Person()
18
19
  cout << "析构函数" << endl;
20
21
  int m_age;
22
  string m_name;
24 };
25
26 void test01()
27 {
  Person p1(10, "lucy"); //构造函数是在实例化对象时会创建,就是在内存开辟空间时
会被调用
   //销毁之前 自动调用析构函数
30
```

```
32 int main()
33 {
34  test01();
35
36  return 0;
37 }
```

3.3 构造函数的分类

无参构造和有参构造 普通沟通和拷贝构造 拷贝构造函数的写法:

```
1 类名(const 类名 &obj){}
```

注意:

- 如果自定义了一个构造函数,系统将不再提供默认的构造函数
- 如果自定义了一个拷贝构造,系统将不再提供默认的拷贝构造
- 默认的拷贝构造是简单的值拷贝
- 如果创建对象时,不能调用对应的构造函数,将不能创建出对象

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <string>
5 using namespace std;
6 class Person
7 {
8 public:
9 //有参和无参构造
10 Person()
11 {
12 cout << "无参构造" << endl;
13
14
15 Person(int a, string n)
16 {
17 cout << "有参构造" << endl;
```

```
18
   age = a;
19
   name = n;
   }
20
   // 拷贝构造的调用时机: 旧对象初始化新对象
21
   //如过自定义了一个拷贝构造,那么系统不再提供默认的拷贝构造
22
   Person(const Person &p)// Person p = p2
23
24
25
   //拷贝构造做了简单的值拷贝
26
   age = p.age;
28
   name = p.name;
   cout << "拷贝构造" << endl;
29
30
   }
31
  int age;
32
   string name;
33
34
35 };
36 void test01()
37 {
  //如果人为提供了一个有参和有参构造,系统将不再提供默认的无参构造
38
  Person p1;//调用无参构造时 不能使用括号法
39
40 Person p2(10, "lucy");
   Person p3(p2);//调用系统提供的默认拷贝构造
41
42
43 }
44 int main()
45 {
  test01();
46
  return 0;
47
48
```

//explicit 关键字 修饰构造函数 作用是不能通过隐式法调用构造函数

```
1 explicit Person(const Person &p){}}
```

```
1 void test02()
2 {
3 //匿名对象 没有名字 生命周期在当前行
```

```
4 Person (10, "lucy");// 调用了有参构造创建了一个匿名对象
5 Person ();//调用了无参构造创建了一个匿名对象
6 Person p1(20, "heihei");
7 //Person (p1);//在定义时匿名对象不能使用括号法调用拷贝构造
8 }
9
10 //显示法调用构造函数
11
12 void test03()
13 {
  Person p1 = Person(10, "lucy");//显示法调用有参构造
14
   Person p2 = Person(p1);//显示法调用拷贝构造
15
  Person p3 = Person();//显示法调用无参构造
16
17
18 }
19
20 // 隐式法调用构造函数
21 void test04()
22 {
  Person p1 = { 10, "lucy" };// 隐式法调用有参构造
23
  Person p2 = p1;// 隐式法调用拷贝构造
  //Person p3 ;// 隐式法不能调用无参构造
25
26
27 }
```

3.4 拷贝构造函数的调用时机

总结一种情况: 旧对象初始化新对象

分类:

- 旧对象初始化新对象
- 形参是一个对象
- 返回局部对象

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <string>
using namespace std;

class Person {
```

```
8 public:
  Person() {
   cout << "no param contructor!" << endl;</pre>
   mAge = 10;
11
   }
12
   Person(int age) {
13
   cout << "param constructor!" << endl;</pre>
14
   mAge = age;
15
   }
16
17
   Person(const Person& person) {
   cout << "copy constructor!" << endl;</pre>
18
   mAge = person.mAge;
19
20
  ~Person() {
21
   cout << "destructor!" << endl;</pre>
   }
23
24 public:
   int mAge;
25
26 };
27 //1. 旧对象初始化新对象
  void test01() {
28
29
   Person p(10);
30
31 Person p1(p);//调用拷贝构造函数
   Person p2 = Person(p);//调用拷贝构造函数
   Person p3 = p; // 相当于Person p2 = Person(p);调用拷贝构造函数
33
  }
34
  //2. 传递的参数是普通对象,函数参数也是普通对象,传递将会调用拷贝构造
  void doBussiness(Person p) {}//Person p = p
38
39 void test02() {
   Person p(10);
40
   doBussiness(p);
41
42 }
43
44 //3. 函数返回局部对象
45 Person MyBusiness() {
   Person p(10);
46
  cout << "局部p:" << (int*)&p << endl;
47
```

```
return p;
49 }
50 void test03() {
51 //vs release、qt下没有调用拷贝构造函数
52 //vs debug下调用一次拷贝构造函数
53 Person p = MyBusiness();
54 cout << "局部p:" << (int*)&p << endl;
55 }
56 int main()
57 {
  test03();
58
  return 0;
59
60 }
61
```

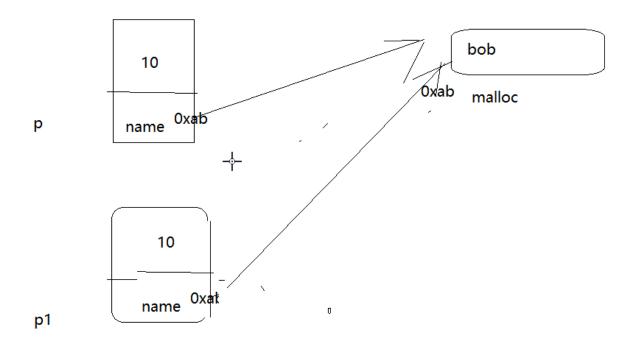
3.5 c++默认增加的函数

默认情况下, c++编译器至少为我们写的类增加3个函数

- 1. 默认构造函数(无参,函数体为空)
- 2. 默认析构函数(无参,函数体为空)
- 3. 默认拷贝构造函数,对类中非静态成员属性简单值拷贝如果用户定义拷贝构造函数, c++不会再提供任何默认构造函数如果用户定义了普通构造(非拷贝), c++不在提供默认无参构造,但是会提供默认拷贝构造

3.6 构造函数的浅拷贝和深拷贝

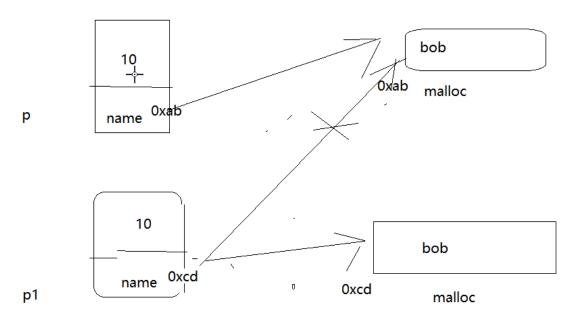
3.6.1 浅拷贝



```
#define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <string>
5 using namespace std;
6 class Person
8 public:
   Person(int age,char *str)
   {
10
   cout << "有参构造" << endl;
11
mAge = age;
name = (char *)malloc(strlen(str)+1);
   strcpy(name, str);
14
   }
15
    void show()
16
17
    cout << name << " " << mAge << endl;</pre>
18
19
20
    ~Person()
21
    if (name != NULL)
22
23
    free(name);
24
    name = NULL;
25
```

```
26
    cout << "析构" << endl;
28
29
    int mAge;
30
    char *name;
31
32
33
34 };
35 void test01()
    Person p(10, "bob");
37
   p.show();
38
   Person p1(p);//调用拷贝构造 函数
39
40
   p1.show();
41
42 }
43 int main()
44 {
   test01();
45
46
   return 0;
47
48 }
```

3.6.2 深拷贝



```
Person(const Person &p)
```

```
2 {
3  mAge = p.mAge;
4  name = (char *)malloc(strlen(p.name)+1);
5  strcpy(name,p.name);
6
7 }
```

3.7多个对象构造和析构

3.7.1 初始化列表

注意:

- 初始化列表,先声明 在调用构造函数时定义并初始化,定义初始化的顺序和声明的顺序一致
- 普通的构造函数,先定义,在赋值

```
1 #define CRT SECURE NO WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <string>
5 using namespace std;
6 class person
7 {
8 public:
   //先定义了int m_a; int m_b; int m_c; 然后在分别赋值
   /*person(int a,int b,int c)
11
   {
12
   m_a = a;
  m_b = b;
13
  m_c = c;
14
   }*/
15
   //先声明了int m_a; int m_b; int m_c 在根据声明的顺序进行定义初始化
16
   //调用构造函数是 定义并初始化,顺序和声明的顺序一致
17
    person(int a, int b, int c) : m_a(a), m_b(b), m_c(c) {} {} // int m_a = a; int m
_c=c;int m_b=b
19
    void show()
20
21
   {
    cout << m_a << " " << m_b << " " << m_c << endl;</pre>
23
    }
   int m a;
```

```
25  int m_c;
26  int m_b;
27  };
28  void test01()
29  {
30
31   person p1(2,3,5);
32   p1.show();
33  }
34  int main()
35  {
36   test01();
37   return 0;
38 }
```

3.7.2 类对象成为另一个类的成员

- 类中有多个对象时,构造的顺序是先构造里面的对象,在构造外面的对象
- 类中有多个对象时,析构时顺序是先析构外面的对象,在析构里面的对象

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <string>
5 using namespace std;
6 class Phone
7 {
8 public:
   Phone(string name)
10 {
  cout << "Phone的构造函数" << endl;
11
   pho_name = name;
12
  }
13
14 ~Phone()
15 {
16 cout << "Phone的析构" << endl;
17
  string pho_name;
```

```
19
20 };
21 class Game
22
   public:
23
    Game(string name)
24
25
    cout << "Game的构造函数" << endl;
26
27
    game_name = name;
28
    ~Game()
29
30
    cout << "Game的析构" << endl;
31
32
33
    string game_name;
34
   class Person
35
36
   public:
37
    /*Person(string per_name1,string pho_name,string g_name)
38
39
40
    per_name = per_name1;
    phone.pho_name = pho_name;
41
    game.game_name = g_name;
42
    }*/
43
    Person(string per_name1, string pho_name, string g_name) :per_name(per_
44
name1), phone(pho_name), game(g_name) {
    cout << "person的构造函数" << endl;
45
46
    void show()
47
48
    cout << per_name << phone.pho_name << " 玩着" << game.game_name << endl;
49
50
51
    ~Person()
    cout << "Person的析构" << endl;
53
54
    string per_name;
55
56
    Game game;
57
    Phone phone;
```

```
58
59 };
60 void test01()
61 {
62
    Person p1("bob","诺基亚","贪吃蛇");
63
   p1.show();
64
65
66
  }
67
68 int main()
69 {
  test01();
71
72
   return 0;
73 }
```

3.8 explicit

C++提供了关键字explicit,禁止通过构造函数进行的隐式转换。声明为 explicit的构造函数不能在隐式转换中使用。

[explicit注意]

explicit用于修饰构造函数,防止隐式转化。

是针对单参数的构造函数(或者除了第一个参数外其余参数都有默认值的多参构造)而言。

3.9 动态对象创建

3.9.1 malloc和free动态申请对象和释放对象

使用malloc和free函数去动态申请对象,和释放申请的对象,不会调用构造函数和析构函数

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
```

```
8 class person
10 public:
person()
  {
12
13 cout << "person无参构造" << endl;
  }
14
15 ~person()
16 {
17 cout << "person析构" << endl;
18
19 int a;
20
21 };
22 void test01()
23 {
24 person *p = (person*)malloc(sizeof(person));
25 free(p);
26 }
27 int main()
29 test01();
30 return 0;
31 }
```

3.9.1 3.9.2 c++中动态申请对象和释放对象

```
类型 *p = new 类型;
delete p;
申请数组:
类型 *p = new 类型[n];
delete []p;
```

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
```

```
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 class person
8 {
9 public:
10
   person()
   {
11
   cout << "无参构造" << endl;
12
13
   ~person()
14
   {
15
   cout << "析构" << endl;
16
17
    }
18
19
    int age;
20
21 };
22 void test01()
23 {
   int *p = new int;//申请一块内存 sizeof(int)大小 并且对这块空间进行初始化
24
25
   cout << *p << endl;</pre>
   *p = 100;
26
  cout << *p << endl;</pre>
27
   delete p;//释放申请的空间
28
29
30
31
33 //申请一个对象
34 void test02()
35 {
    person *p = new person;//sizeof(person)
36
    delete p;
37
38
   }
39
40 //申请一个数组
41 void test03()
42 {
43 //new一个数组时,返回的是该数组的首元素的地址
```

```
44 int *p = new int[10];
45 for (int i = 0; i < 10; i++)
46
    p[i] = i + 100;
47
48
   for (int i = 0; i < 10; i++)
49
50
  cout << p[i] << " ";
51
52
53 cout << endl;</pre>
54 delete []p;
55
56 }
57 int main()
58 {
59
  test03();
60 return 0;
61 }
```

3.9.3 用于数组的new和delete

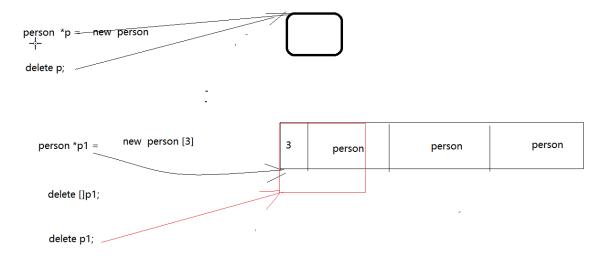
```
1 void test04()
2 {
3     //new时调用有参构造
4     person *p = new person(10);
5     delete p;
6     person *p1 = new person[10];//注意new对象的数组时不能调用有参构造 只能调用无参构造
7     delete []p1;
8
9 }
```

3.9.4 delete void *问题

```
1 void test05()
2 {
3  void *p = new person;
4  delete p;// p的类型是void *所有不会调用析构函数
5 }
```

3.9.5 使用new和delete采用相同形式

- new单一对象时,使用delete释放单一的对象
- new一个数组时,使用delete []释放这个数组

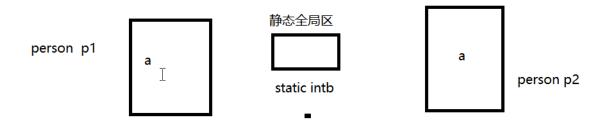


3.10 静态成员

在类定义中,它的成员(包括成员变量和成员函数),这些成员可以用 关键字static声明为静态的,称为静态成员。

不管这个类创建了多少个对象,静态成员只有一个拷贝,这个拷贝被所 有属于这个类的对象共享。

3.10.1 静态成员变量



- 静态成员变量在内存中只有一份,多个对象共享一个静态变量
- 静态成员变量,必须类内声明,类外定义
- 静态成员变量可以通过类的作用域访问
- 静态成员变量可以通过类的对象访问

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
```

```
5 #include <string>
6 using namespace std;
7 class person
8 {
9 public:
10 int a;
11 //静态成员变量不能再类内初始化 类内只能声明,定义在全局 声明的作用只是限制静态
成员变量作用域
  static int b;//静态成员变量 在编译阶段就分配内存 存在静态全局区
13 };
14 int person::b = 10;//类中成员变量的定义
15 void test01()
16 {
17 person p1;
18 p1.b = 100;
19
  cout << p1.b << endl;</pre>
20
21 }
22 void test02()
23 {
  cout << person::b << endl;//通过类的作用域访问类的静态成员函数
   //cout << person::a << endl;</pre>
25
26
27 }
28
29 using namespace std;
30 int main()
31 {
   test02();
32
33
34
  return 0;
35 }
```

3.10.2 静态成员函数

- 静态成员函数能访问静态成员变量不能访问普通的成员变量
- 可以通过类的作用域访问静态成员函数
- 可以通过对象访问静态成员函数

```
1 #define _CRT_SECURE_NO_WARNINGS
```

```
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 class person
8 {
9 public:
   int a;
  //静态成员变量不能再类内初始化 类内只能声明,定义在全局 声明的作用只是限制静态
成员变量作用域
12 static int b;//静态成员变量 在编译阶段就分配内存 存在静态全局区
  void show()
13
14 {
15  cout << a << " " << b << endl;;</pre>
17 static void static_show()//静态成员函数 可以访问静态成员变量 不能访问普通的
成员变量
18
  {
  cout << " " << b << endl;;
19
  }
20
21
22 };
23 int person::b = 100;
24 void test01()
25 {
   person::static_show();//通过流类的作用域访问静态成员函数
27 person p1;
  p1.static show();//通过对象访问静态成员函数
29
  }
30
31 int main()
32 {
  test01();
33
34 return 0;
35 }
```

3.10.3 const修饰的静态成员变量

- const修饰的静态成员变量保存在常量区,只读的,在内存中只有一份
- const修饰的静态成员变量可以在类内定义且初始化
- const修饰的静态成员变量可以通过类的作用域访问
- const修饰的静态成员变量可以通过对象访问
- 静态成员函数可以访问const修饰的静态成员变量

```
1 const int num = 10;//const修饰的全局变量保存在常量区 不可更改
2 class person
3 {
4 public:
5 int a;
6 //静态成员变量不能再类内初始化 类内只能声明,定义在全局 声明的作用只是限制静态成
员变量作用域
7 static int b;//静态成员变量 在编译阶段就分配内存 存在静态全局区
8 const static int c=1000;//const修饰的静态成员变量 是保存在常量区 不可修改(只
读) 在内存中只有一份
9 };
10 int person::b = 10;//类中成员变量的定义
11
12 void test()
13 {
14 cout << person::c << endl;</pre>
person p1;
16 cout << p1.c << endl;</pre>
17
18 }
```

3.10.4 单例模式

单例模式: 一个类只能创建出一个对象

单例模式实现的步骤:

- 1. 将无参构造私有化
- 2. 将拷贝构造私有化
- 3. 定义一个静态的成员指针变量指向new出来的一个唯一对象

4. 将静态的成员指针变量私有化,提供获得唯一对象的地址的静态接口

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 class Feifei
8 {
9 public:
   int age;
10
int yanzhi;
12 static Feifei * instance()
13 {
14 return single;
15 }
16 private:
  static Feifei *single;
17
18
   Feifei() //无参构造私有化
   {}
19
  Feifei(const Feifei &p)
20
21 {}
22
23
24 };
25 Feifei * Feifei::single = new Feifei;
27 void test03()
28 {
   Feifei * p = Feifei::instance();
29
    p \rightarrow age = 10;
30
    p->yanzhi = 20;
31
32
   Feifei * p1 = Feifei::instance();
33
   cout << p1->age << " " << p1->yanzhi << endl;</pre>
34
```

```
36
37
38
39 void test02()
   //Feifei::single->age = 100;
41
   //Feifei::single->yanzhi = 100;
42
43
  //Feifei p1(*Feifei::single);//调用拷贝构造实例化出一个对象
44
   // Feifei::single = NULL;
45
46
47 }
48 void test01()
49 {
   //Feifei p1;//需要调用无参构造
50
  //Feifei p2;
51
52
53 }
54 int main()
55 {
  test03();
56
  return 0;
58 }
```

四. 类对象成员的初探

1.成员变量和函数的存储

- 类对象成员-普通成员变量占用对象空间大小
- 类对象成员-静态成员变量不占用对象空间大小
- 类对象成员-普通成员函数不占用对象空间大小
- 类对象成员-静态成员函数不占用对象空间大小

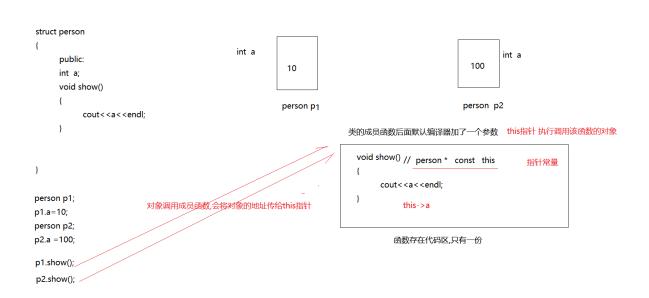
```
class person
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 静态全局区
  public:
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      static int b;
                                                               : \inf_{a,\cdot}/普通的成员变量 \inf_{a,\cdot}/普通的成员变量 \inf_{a,\cdot} \inf_{a,
                                                                                                                               cout << a << " " << b << endl;
                                                                       ,
static void show1()//静态成员函数 不存在类实例化的对象中
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           int a
                                                                                                                                  cout << b << end1;
};
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        void show()//普通成员函数不存在类实例化的对象中
       person p;
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          cout << a << " " << b << endl;
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            ,
static void show1()//静态成员函数 不存在类实例化的对象中
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                代码区
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          cout << b << end1;
```

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 class person
8
9 public:
    int a;//普通的成员变量
    static int b;//静态成员不存在类实例化的对象中
11
    void show()//普通成员函数不存在类实例化的对象中
13
   cout << a << " " << b << endl;
14
15
    static void show1()//静态成员函数 不存在类实例化的对象中
16
17
   cout << b << endl;</pre>
18
19
20
   };
   int person::b = 1;
   void test01()
22
23
24
    person p;
25
    p.show();
    //空类的大小不是0 而是1
26
    cout << sizeof(person) << endl;</pre>
27
28
29
30 int main()
```

```
31 {
32 test01();
33
34 }
```

2 this指针的工作原理

• 类的成员函数默认编译器都加上了一个this指针,这个this指针 指向调用该成员函数的对象



3this指针的应用

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 class person
8 {
9 public:
   person(int age, string name)// this
11
   this->age = age;
12
   this->name = name;
13
14
   void show()
```

```
16
    cout << age << " " << name << endl;</pre>
18
    person person_add( person &p2)//this ----> p1
19
20
    person p(this->age+p2.age,this->name+p2.name);//"helloworld"
21
22
    return p;
23
    }
24
    int age;
25
    string name;
26
27
28
  };
   person person_add(person &p1, person &p2)
30 {
    person p(p1.age+p2.age,p1.name+p2.name);//"helloworld"
31
    return p;
33 }
34 void test02()
35
36
    person p1(10, "hello");
37
    person p2(20, "world");
    //p3 = p1 + p2 30, "helloworld"
38
    //person p3 = person_add(p1,p2);
39
   //p3.show();
40
    person p3 = p1.person_add(p2);
41
    p3.show();
42
43
44
45
47 void test01()
48 {
    person p1(10,"lucy");
49
   p1.show();
50
51
52
   }
53 int main()
54 {
    test02();
55
56
```

```
57 return 0;
58 }
```

4 const修饰的成员函数

- 在函数后面加上const,这个是一个常函数
- 这个const修饰的是指针 const type * const this,代表不能 通过this指针去修改this指针指向对象的内容

```
1 //常函数 不能通过this指针修改this指针指向的对象内容
2 //常量指针常量
3 person person_add( person &p2)const//const person * const this -----> p
1
4 {
5 //this->age = 200;
6 person p(this->age+p2.age,this->name+p2.name);//"helloworld"
7 return p;
8 }
```

五:友元

类的主要特点之一是数据隐藏,即类的私有成员无法在类的外部(作用域之外)访问。但是,有时候需要在类的外部访问类的私有成员,怎么办?

解决方法是使用友元函数,友元函数是一种特权函数,**c++**允许这个特权函数访问私有成员。这一点从现实生活中也可以很好的理解:

比如你的家,有客厅,有你的卧室,那么你的客厅是Public的,所有来的客人都可以进去,但是你的卧室是私有的,也就是说只有你能进去,但是呢,你也可以允许你的闺蜜好基友进去。

如果想要让全局函数或一个类的成员函数访问另一个类私有成员,只需要声明友元即可

1 友元的语法

1.1全局函数成为类的友元

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 class Building
8 {
  friend void print_Building(Building &b);
10 public:
   Building(string hall,string bedroom)
11
12
13 this->bedroom = bedroom;
14 this->hall = hall;
15
16 string hall;
17 private:
   string bedroom;
19
20 };
21
22 void print Building(Building &b)
23 {
   cout << b.hall << " " << b.bedroom << endl;</pre>
24
25 }
27 void test01()
28 {
   Building b1("凌霄殿","闺房");
29
   print_Building(b1);
31 }
32 int main()
33 {
34 test01();
   return 0;
```

1.2 类成为另一个类的友元,类的成员函数成为另一个类的友元

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
```

```
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
8 class Building;
9 class Goodgay
10 {
11 public:
12
   Goodgay(string hall, string bedroom);
13
  void visit();
14
   Building *b;
15
16 };
17 class Building
    //friend void print_Building(Building &b);
19
   //friend class Goodgay; //一个类成为另一个类的友元
20
   friend void Goodgay::visit();//类的成员函数成为另一类的友元
21
22 public:
    Building(string hall, string bedroom)
23
24
   this->bedroom = bedroom;
25
   this->hall = hall;
26
27
   string hall;
28
29 private:
    string bedroom;
30
31
32 };
33
   Goodgay::Goodgay(string hall, string bedroom)
34
35 {
   b = new Building( hall, bedroom);
37 }
38
39 void Goodgay::visit()
40 {
    cout << b->hall << " " << b->bedroom << endl;</pre>
41
```

```
43 void test01()
44 {
45 Goodgay gd("凌霄殿","闺房");
46 gd.visit();
47
48 }
49 int main()
50 {
51 test01();
52
53 return 0;
54 }
55
56
```

六: 运算符重载

1 运算符重载的基本概念

```
运算符重载: 就是给运算符赋予一个新的意义 int a = 1; int b = 2;' int c = a +b; person p1; person p2; person p3 = p1+p2;
```

运算符只能运算内置的数据类型,对于自定义的数据类型,不能运算,所以 我们可以重载运算符

```
类的成员函数实现
                                                 全局函数实现类的对象相加
I class person
                                                                                             person person_add(person &p2)
                                                person person_add(person &p1, person &p2)
   public:
                                                                                                  person p(this->age+p2.age);
        person(int age)
                                                     person p(p1.age+p2.age);
             this->age =age;
                                                     return p;
        int age;
   };
  person p1(10);
  person p2(20);
  //p3.age = P1.age + p2.age
                                                 operator全局函数
                                                                                                 写一个operatorc成员函数
 person p3 = person_add(p1, p2);
                                              person operator+ (person &p1, person &p2)
 person p3 = p1.person_add(p2);
                                                                                                     person p(this->age+p2.age);
 person p3 = p1 + p2;
                                                   person p(pl.age+p2.age);
                                                                                                     return p;
                        operator+(p1, p2)
                                                   return p:
                        pl. operator+(p2)
```

2 重载加号运算符

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
  class person
8
  public:
10
    person(int age)
11
12
    this->age = age;
13
14
    person operator+( person &p2)
15
16
17
    person p(this->age+p2.age);
18
    return p;
19
    }
20
21
    int age;
22
23
   };
24
   //person operator+(person &p1, person &p2)
26
  //{
```

```
27 // person p(p1.age+p2.age);
28 // return p;
29 //}
30
31 void test01()
32
33
   person p1(10);
34 person p2(20);
   person p3 = p1 + p2;// operator+(p1,p2) p1.operator+(p2)
35
  cout << p3.age << endl;</pre>
37 }
38
39
40 int main()
41 {
   test01();
42
43 return 0;
44 }
45
```

3 重载左移运算符和算符重载碰上友元函数

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 class person
9 friend ostream& operator<<(ostream &cout, person &p);</pre>
10 public:
  person(int age)
11
12
   this->age = age;
13
   }
14
15 private:
   int age;
17
18 };
```

```
19 ostream& operator<<(ostream &cout, person &p)
20 {
21 cout << p.age;</pre>
  return cout;
22
23 }
24 void test01()
25 {
26 person p1(10);
27 cout << p1 <<endl;</pre>
   // operator<<(cout,p1) //cout.operator<<(p1)</pre>
28
29
30 }
31 int main()
32 {
33
   test01();
34
   return 0;
35
36 }
```

4 可以重载的运算符

几乎C中所有的运算符都可以重载,但运算符重载的使用时相当受限制的。特别是不能使用C中当前没有意义的运算符(例如用**求幂)不能改变运算符优先级,不能改变运算符的参数个数。这样的限制有意义,否则,所有这些行为产生的运算符只会混淆而不是澄清寓语意。

可以重载的操作符

不能重载的算符

. :: .* ?: sizeof

5 重载自加自减运算符

++a ; 先自加 在使用

a++;//先使用 在自加

- 前置加加返回的是引用
- 后置加加返回的是对象
- 前置加加调用TYPE& operator++()函数
- 后再加加调用的是TYPE operator++(int)函数,也就是后置加加多了一个占位参数

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 class Myint
8 {
9 public:
  Myint(int num)//this
11
  this->num = num;
12
13
  Myint& operator++()
14
15 {
```

```
this->num = this->num + 1;
    return *this;
17
18
    Myint operator++(int)
19
20
    Myint tmp = *this;
21
22
   //加加
   this->num = this->num + 1;
23
24
   return tmp;
25
26
   int num;
27
28 };
   ostream & operator<<(ostream &cout, Myint &p)
30 {
31
   cout << p.num;</pre>
32
   return cout;
33
34 }
35 void test01()
36 {
37
   Myint p1(10);
38 cout << p1 << endl;</pre>
  ++p1;//operator++(p1) p1.operator++()
39
40 cout << ++p1 << endl;
  cout << p1++ << endl;//p1.operator++(int)</pre>
42 }
43 int main()
44 {
   test01();
45
46
   return 0;
47
```

6 智能指针

我们经常new一个对象,忘记释放,所以我们使用智能指针来维护智能指针实质是一个局部对象,这个局部对象维护了new出来的对象的地址,在

局部对象的析构函数中,会帮忙释放new出来的对象 对于智能指针我们重载了-> 和* 让指针指针和普通指针一样使用

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 class person
8 {
9 public:
10 person(int age)
11 {
12 this->age = age;
   }
13
   int age;
14
15 };
16 class Smartpointer
17 {
18 public:
   Smartpointer(person *p1)
19
20
  this->p = p1;
21
22
   ~Smartpointer()
23
  {
24
   delete p;
25
26
   cout << "释放了p" << endl;
27
    person *operator->()
28
29
    return p;
30
31
    person& operator*()
32
33
    return *p;
34
35
    }
    person *p;
37 };
```

```
38 void test01()
  {
39
   // 局部对象 在释放之前可以帮助释放 p,
40
  //person *p = new person(10);
41
   Smartpointer sp(new person(10));
42
   //cout << p->age << endl;</pre>
44 cout << sp->age << endl;//sp-> 返回的是p p sp.operator->()
  cout << (*sp).age << endl;//sp.operator*()</pre>
45
  //忘记释放p指向申请的对象
46
47 }
48 int main()
49
  {
  test01();
50
  return 0;
51
52 }
```

7 重载=号运算符

编译器默认给每一个类加上了4个函数

- 默认无参构造
- 默认的拷贝构造
- 析构
- operator=()

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 class person
8 {
9 public:
10 person() {}
   person(int age1,char *name1)
11
   {
12
   age = age1;
13
   name = new char[strlen(name1) + 1];
15 strcpy(name,name1);
```

```
16
    person& operator=(person &p1)//this- ..> p2
18
   this->age = p1.age;
19
    this->name = new char[strlen(p1.name)+1];
20
    strcpy(this->name, p1.name);
21
22
    return *this;
23
24
    ~person()
25
26
    delete []name;
27
28
   int age;
29
30
   char *name;
31
32
   };
33 void test01()
34 {
   person p1(10, "bob");
35
36 person p2;
  p2 = p1;//p2.operator=(p1)
   cout << p2.age << " " << p2.name << endl;</pre>
38
39
40 }
41 int main()
42 {
43 test01();
44 return 0;
45 }
```

8 重载等于和不等于号

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class person

{
```

```
9 public:
    person(int age, string name)
11
   this->age = age;
12
    this->name = name;
13
14
15
    bool operator==(person &p2)//this->>>p1
16
    return this->age == p2.age && this->name == p2.name;
17
18
    bool operator!=(person &p2)//this->>>p1
19
20
    return this->age != p2.age || this->name != p2.name;
21
22
   int age;
23
   string name;
24
25 };
26 void test01()
27 {
28
   person p1(10, "lucy");
    person p2(20, "lucy");
29
    if (p1 == p2)//p1.operator==(p2)
30
31
    cout << "p1 == p2" << endl;</pre>
32
33
    if (p1 != p2)//p1.operator!=(p2)
35
    cout << "p1 != p2" << endl;</pre>
36
37
38
39
40 }
41 int main()
42 {
43 test01();
44 return 0;
45 }
```

一个类中重载了()的类,那么整个类定义处来的对象可以像函数一样使用,本质是调用了operator()整个函数

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 class Myadd
8 {
9 public:
10
    int add(int a, int b)
11
12
   return a + b;
13
14
   }
   int operator()(int x, int y)
15
  {
16
   return x+ y;
   }
18
19
  };
20
21 void test()
22 {
  //Myadd p;
  //cout << p.add(3, 4) << endl;
24
  // p() 可以像函数一样调用的对象 函数对象
25
  //cout << p(3, 4) << endl;//p.operator()(3,4)</pre>
26
   cout << Myadd ()(3, 4) << endl;//匿名对象 Myadd ().operator()(3,4)
28
29
30 int main()
31 {
   test();
32
   return 0;
34
35 }
```

不能重载operator&& 和 operator|| 的原因是,无法在这两种情况下实现内置操作符的完整语义。说得更具体一些,内置版本版本特殊之处在于: 内置版本的&&和||首先计算左边的表达式,如果这完全能够决定结果,就无需计算右边的表达式了--而且能够保证不需要。我们都已经习惯这种方便的特性了。

我们说操作符重载其实是另一种形式的函数调用而已,对于函数调用总是在函数执行之前对所有参数进行求值。

```
1 class Complex{
2 public:
3 Complex(int flag){
4 this->flag = flag;
5 }
6 Complex& operator+=(Complex& complex){
7 this->flag = this->flag + complex.flag;
8 return *this;
9 }
10 bool operator&&(Complex& complex){
11 return this->flag && complex.flag;
12 }
13 public:
14 int flag;
15 };
16 int main(){
17
  Complex complex1(0); //flag 0
  Complex complex2(1); //flag 1
19
20
21 //原来情况,应该从左往右运算,左边为假,则退出运算,结果为假
22 //这边却是,先运算(complex1+complex2),导致,complex1的flag变为complex1+c
omplex2的值, complex1.a = 1
23 // 1 && 1
24 //complex1.operator&&(complex1.operator+=(complex2))
  if (complex1 && (complex1 += complex2)){
          //complex1.operator+=(complex2)
26
27 cout << "真!" << endl;
29 else{
```

```
30 cout << "假!" << endl;
31 }
32
33 return EXIT_SUCCESS;
34 }
35
```

11 重载运算符建议

=, [], () 和 -> 操作符只能通过成员函数进行重载 << 和 >>只能通过全局函数配合友元函数进行重载 不要重载 && 和 || 操作符,因为无法实现短路规则常规建议

运算符	建议使用
所有的一元运算符	成员
= () [] -> ->*	必须是成员
+= -= /= *= ^= &= != %= >>= <<=	成员
其它二员运算符	非成员

12 封装string类

```
1 MyString.h
2 #define _CRT_SECURE_NO_WARNINGS
3 #pragma once
4 #include <iostream>
5 using namespace std;
7 class MyString
9 friend ostream& operator<< (ostream & out, MyString& str);</pre>
10 friend istream& operator>>(istream& in, MyString& str);
11
12 public:
13 MyString(const char *);
14 MyString(const MyString&);
15 ~MyString();
16
17 char& operator[](int index); //[]重载
18
```

```
19 //=号重载
20 MyString& operator=(const char * str);
21 MyString& operator=(const MyString& str);
22
  //字符串拼接 重载+号
24 MyString operator+(const char * str );
25 MyString operator+(const MyString& str);
26
  //字符串比较
27
28 bool operator== (const char * str);
29 bool operator== (const MyString& str);
30 private:
31 char * pString; //指向堆区空间
32 int m_Size; //字符串长度 不算'\0'
33 };
34 MyString.cpp
35 #include "MyString.h"
36
37 //左移运算符
38 ostream& operator<< (ostream & out, MyString& str)
39 {
40 out << str.pString;
41 return out;
42 }
43 // 右移运算符
44 istream& operator>>(istream& in, MyString& str)
45 {
46 //先将原有的数据释放
47 if (str.pString != NULL)
48 {
49 delete[] str.pString;
50 str.pString = NULL;
51
  char buf[1024]; //开辟临时的字符数组,保存用户输入内容
  in >> buf;
53
54
  str.pString = new char[strlen(buf) + 1];
56 strcpy(str.pString, buf);
  str.m_Size = strlen(buf);
58
```

```
59 return in;
60 }
62 //构造函数
63 MyString::MyString(const char * str)
64 {
65 this->pString = new char[strlen(str) + 1];
66 strcpy(this->pString, str);
67 this->m_Size = strlen(str);
68 }
69
70 //拷贝构造
71 MyString::MyString(const MyString& str)
72 {
73 this->pString = new char[strlen(str.pString) + 1];
74 strcpy(this->pString, str.pString);
75 this->m_Size = str.m_Size;
76 }
77 //析构函数
78 MyString::~MyString()
79 {
80 if (this->pString!=NULL)
81 {
82 delete[]this->pString;
83 this->pString = NULL;
84 }
85 }
86
87 char& MyString::operator[](int index)
88 {
89 return this->pString[index];
   }
90
91
92 MyString& MyString::operator=(const char * str)
93 {
94 if (this->pString != NULL){
95 delete[] this->pString;
96 this->pString = NULL;
97 }
98 this->pString = new char[strlen(str) + 1];
```

```
99 strcpy(this->pString, str);
100 this->m_Size = strlen(str);
101 return *this;
102 }
103
104 MyString& MyString::operator=(const MyString& str)
105 {
106 if (this->pString != NULL){
107 delete[] this->pString;
108 this->pString = NULL;
109 }
110 this->pString = new char[strlen(str.pString) + 1];
111 strcpy(this->pString, str.pString);
112 this->m_Size = str.m_Size;
113 return *this;
114 }
115
116
117 MyString MyString::operator+(const char * str)
118 {
int newsize = this->m Size + strlen(str) + 1;
120 char *temp = new char[newsize];
121 memset(temp, 0, newsize);
122 strcat(temp, this->pString);
123 strcat(temp, str);
124
125 MyString newstring(temp);
126 delete[] temp;
127
128 return newstring;
129 }
130
131 MyString MyString::operator+(const MyString& str)
132 {
int newsize = this->m_Size + str.m_Size + 1;
134 char *temp = new char[newsize];
135 memset(temp, 0, newsize);
136 strcat(temp, this->pString);
137 strcat(temp, str.pString);
138
139 MyString newstring(temp);
```

```
140 delete[] temp;
141 return newstring;
142 }
143
144 bool MyString::operator==(const char * str)
if (strcmp(this->pString, str) == 0 && strlen(str) == this->m_Size){
147 return true;
148 }
149
150 return false;
151 }
152
153 bool MyString::operator==(const MyString& str)
if (strcmp(this->pString, str.pString) == 0 && str.m_Size == this->m_Siz
e){
156 return true;
157 }
158
159 return false;
160 }
161 TestMyString.cpp
162 void test01()
163 {
164 MyString str("hello World");
165
166 cout << str << endl;</pre>
167
168 //cout << "请输入MyString类型字符串: " << endl;
169 //cin >> str;
170
171 //cout << "字符串为: " << str << endl;
172
173 //测试[]
174 cout << "MyString的第一个字符为: " << str[0] << endl;
175
176 //测试 =
177 MyString str2 = "^_^";
178 MyString str3 = "";
179 str3 = "aaaa";
```

```
180 str3 = str2;
181 cout << "str2 = " << str2 << endl;</pre>
182 cout << "str3 = " << str3 << endl;</pre>
183
184 //测试 +
185 MyString str4 = "我爱";
186 MyString str5 = "北京";
187 MyString str6 = str4 + str5;
188 MyString str7 = str6 + "天安门";
189
190 cout << str7 << endl;
191
192 //测试 ==
193 if (str6 == str7)
194 {
195 cout << "s6 与 s7相等" << endl;
196 }
197 else
198 {
199 cout << "s6 与 s7不相等" << endl;
200 }
201 }
```

13 优先级

运算符和结合性

优先级。	运算符	名称或含义。	使用形式。	结合方向。	说明。
	0.	数组下标。	数组名[常量表达式]。		₀
	() -	圆括号。	(表达式)/函数名(形参表)。	+ 51+	₄
1.	. 0	成员选择(对象)。	对象.成员名。	左到右。	»
	-> .	成员选择(指针)。	对象指针->成员名。		,
P					
	• 0	负号运算符。	-表达式。	右到左。	
	~ .	按位取反运算符。	~表达式。		
2 0	++ .	自增运算符。	++变量名/变量名++。		单目运算符。
	 ,	自减运算符。	变量名/变量名。		
	* ,	取值运算符。	*指针变量。		

				_	
	& .	取地址运算符。	&变量名。		
	! 0	逻辑非运算符。	!表达式。		
	(类型)	强制类型转换。	(数据类型)表达式。		
	sizeof	长度运算符。	sizeof(表达式)。		,
٥					
	1.	除。	表达式/表达式。		
3 -	* ,	乘。	表达式*表达式。	左到右。	双目运算符。
	% ₽	余数(取模)。	整型表达式%整型表达式。		
	+ .	加。	表达式+表达式。	de Wilde	
4.	• o	减 。	表达式-表达式。	左到右。	双目运算符。
	<<.	左移。	变量<<表达式。		
5.	>> .	右移。	变量>>表达式。	左到右。	双目运算符。

6.	>。 大于	表达	式>表达式。	左到右。	双目运算符。
	>= 。 大于	等于。 表达	式>=表达式。		
	<。 小于	表达	式<表达式。		
	<=。 小于	等于, 表达	式<=表达式。		
7.	==。 等于	表达	式==表达式。	4.714	右。 双目运算符
	!=。不等	表达	式!= 表达式。	左到右。	

8 -	& .	按位与。	表达式&表达式。	左到右。	双目运算符。
9	A ₂	按位异或,	表达式^表达式。	左到右。	双目运算符。
10 -	P	按位或。	表达式 表达式。	左到右。	双目运算符。
11 -	&&.	逻辑与。	表达式&&表达式。	左到右。	双目运算符。
12 -		逻辑或。	表达式 表达式。	左到右。	双目运算符。
o					
13 -	?:.	条件运算符。	表达式 1?。 表达式 2: 表达式 3。	右到左。	三目运算符。

		赋值运算符。	变量=表达式。	-	,
	/= .	除后赋值。	变量/=表达式。		0
14 -	*= .	乘后赋值。	变量*=表达式。	右到左。	,
	%= -	取模后赋值。	变量%=表达式。		₀
	+= .	加后赋值。	变量+=表达式。		0

	-= ,	减后赋值。	变量-=表达式。		a
	<<= ,	左移后赋值。	变量<<=表达式。		₄
	>>= ,	右移后赋值。	变量>>=表达式。		₄
	&=.	按位与后赋值。	变量&=表达式。		 ,
	^= .	按位异或后赋值。	变量^=表达式。		,
	= 0	按位或后赋值。	变量 =表达式。		,
ē					
15 -	, o	逗号运算符。	表达式,表达式,	左到右。	,

七.继承

1继承的概念

1.1 为什么需要继承

一个类继承另一个类,这样类中可以少定义一些成员

```
//person(个人)类

class person
{
    private:
        char name[10];
        int age;
        char sex;

public:
        void print();
};
```

直接定义employee类,代码重复比较严重。

1.2 继承的概念

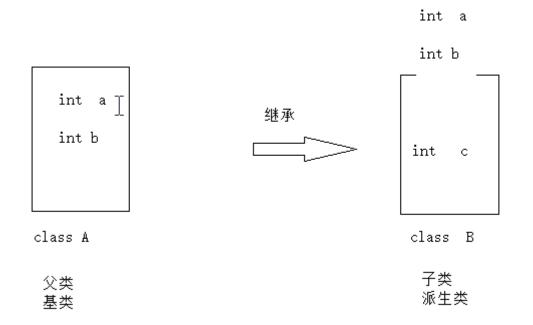
C++最重要的特征是代码重用,通过继承机制可以利用已有的数据类型 来定义新的数据类型,新的类不仅拥有旧类的成员,还拥有新定义的成员。

一个B类继承于A类,或称从类A派生类B。这样的话,类A成为基类(父 类), 类B成为派生类(子类)。

派生类中的成员,包含两大部分:

一类是从基类继承过来的,一类是自己增加的成员。

从基类继承过过来的表现其共性,而新增的成员体现了其个性。



1.3 派生类的定方义法

派生类定义格式:

Class 派生类名: 继承方式 基类名{

//派生类新增的数据成员和成员函数

}

三种继承方式:

public: 公有继承 private: 私有继承

protected: 保护继承

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 // 基类
8 class Animal
9 {
10 public:
int age;
  void print()
12
13
  {
   cout << age << endl;</pre>
14
```

```
16 };
17 class Dog :public Animal
18 {
19 public:
20 int tail_len;
21
22 /*
23 int age;
24 void print()
25 {
26 cout << age << endl;</pre>
27 }
28 */
29 };
31 void test01()
33 Dog d;
34
35
36 }
37
38 int main()
39 {
40
41 return 0;
```

2 派生类访问权限控制

```
class A
         I
                                public:
                                       int a;
                                protected:
                                       int b;
                                private:
                                       int c;
            公有继承
                                                            私有继承
                             保护继承
                              class B: protected A
class B: public A
                                                                    class B :private A
                              protected:
public:
                                                                    private:
                                     int a;
       int a;
                                                                          int a;
                                     int b;
protected:
                                                                          int b;
                              不可访问:
       int b;
                                                                    不可访问:
                                     int c;
不可访问:
                                                                          int c;
                              };
       int c;
                                                                    };
```

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 class Base
9 public:
10 int a;
11 protected:
12
  int b;
13 private:
  int c;
14
15 };
16 //公有的继承方式 基类中是什么控制权限,继承到子类中也是什么控制权限
17 class A :public Base
18 {
  /*
19
  public:
20
   int a;
```

```
22 protected:
23 int b;
24 private:
  int c;
25
  */
26
27 public:
28
   int d;
  void show()
29
30
  {
  //子类的成员函数去访问父类的成员 子类不可以访父类的私有成员
31
  //cout << a << b << c << endl;
32
33
  }
34
35 };
36 class B :protected Base
37 {
38 public:
39 /*
40 //公有继承 将父类中的公有的权限变成保护的,其他不变
41 protected:
42 int a;
43 protected:
44 int b;
45 private:
46
  int c;
   */
47
48
49
  int d;
50
  void show()
51
52
  //子类访问父类 不能访问父类的私有成员
53
  //cout << a << b << c << d << endl;
54
  }
55
56 };
57
  class C : private Base
58
59
  //私有继承 会将所有的权限都变成私有的
60
61 /*
```

```
62
  private:
  int a;
  private:
64
   int b;
  private:
66
   int c;
67
68
   */
69 public:
  int d;
70
  void show()
71
72
  //cout << a << b << c << d << endl;
73
74 }
75
76 };
77 void test01()
78 {
79 A p;
  //p通过类外可以访问公有的权限
80
81
  p.a = 10;
  p.d = 20;
82
   B p1;
83
  //p1.a = 100;
84
   p1.d = 200;
85
86
  C p2;
87
  //p2.a = 100;
88
   p2.d = 100;
89
90
91 }
92
93
94 int main()
95 {
96
97 return 0;
98 }
```

3 继承中的析构和构造

3.1 继承中的对象模型

在C++编译器的内部可以理解为结构体,子类是由父类成员叠加子类新成员而成:

```
1 class Aclass{
2 public:
3 int mA;
4 int mB;
5 };
6 class Bclass : public Aclass{
7 public:
8 int mC;
9 };
10 class Cclass : public Bclass{
11 public:
12 int mD;
13 };
14 void test(){
15 cout << "A size:" << sizeof(Aclass) << endl;//8</pre>
16 cout << "B size:" << sizeof(Bclass) << endl;//12</pre>
17 cout << "C size:" << sizeof(Cclass) << endl;//16
18 }
```

3.2 对象构造和析构的调用原则

继承中的构造和析构

- 子类对象在创建时会首先调用父类的构造函数,父类构造函数执行完毕后,才会调用子类的构造函数
- 当父类构造函数有参数时,需要在子类初始化列表(参数列表)中显示调用父类构造函数
- 析构函数调用顺序和构造函数相反

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class Base

{
public:
Base(int age, string name)
```

```
11 {
12 this->age = age;
  this->name = name;
13
  cout << "Base的构造函数" << endl;
14
   }
15
   ~Base()
16
   {
17
   cout << "Base的析构函数" << endl;
18
19
   }
20
   int age;
   string name;
21
22
23 };
24 //创建子类对象时,必须先构建父类 需要调用父类的构造函数
25 class Son:public Base
26 {
27 public:
   Son(int id,int age,string name):Base(age, name)
28
   {
29
  this->id = id;
30
  cout << "Son 的构造函数" << endl;
31
32
   }
   ~Son()
  {
34
   cout << "Son 的析构函数" << endl;
   }
36
37
   int id;
38
39 };
40 void test01()
41 {
42
  Son p(10,18,"lucy");
43
44 }
45 int main()
46 {
47 test01();
   return 0;
49 }
50
```

4.继承中同名成员的处理问题

如果子类和父类有同名的成员变量和成员函数,发送继承时,父类的成员变量和成员函数会被隐藏

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 class Base
8 {
9 public:
10 Base(int a)
  {
11
  this->a = a;
12
13 }
14 void foo()
15 {
16 cout << "父类的foo函数" << endl;
  int a;
18
19 };
20 class Son :public Base
21 {
22 public:
   Son(int a1, int a2):Base(a1),a(a2)
23
25 void foo()
26
  cout << "子类的foo函数" << endl;
27
   int a;
29
30
31 };
32 void test01()
33 {
   Son p(10, 20);
34
  //如果子类和父类有同名的成员变量,父类的成员变量会被隐藏,访问的是子类的成员变量
```

```
      36  //如果子类和父类有同名的成员函数,父类的成员函数会被隐藏,访问的是子类的成员函数

      37  cout << p.a << endl;</td>

      38  p.foo();

      39  40 }

      41  int main()

      42  {

      43  test01();

      44  return 0;

      45 }
```

5 非自动继承的函数

发送继承时,子类不会继承父类的构造函数,析构函数和operator=函数

6 继承中的静态成员特性

发送继承时,子类和父类有同名的静态成员函数或静态成员变量.父类中的静态成员函数或静态成员变量会被隐藏

```
1 #define CRT SECURE NO WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
8 class Base {
9 public:
10 static int getNum() { return sNum; }
static int getNum(int param) {
12 return sNum + param;
13 }
14 public:
15 static int sNum;
16 };
17 int Base::sNum = 10;
18
19 class Derived : public Base {
20 public:
21 static int sNum; //基类静态成员属性将被隐藏
22 #if 0
```

```
//重定义一个函数,基类中重载的函数被隐藏
23
  static int getNum(int param1, int param2) {
24
  return sNum + param1 + param2;
25
  }
26
27 #else
  //改变基类函数的某个特征,返回值或者参数个数,将会隐藏基类重载的函数
28
  static void getNum(int param1, int param2) {
29
30 cout << sNum + param1 + param2 << endl;</pre>
32 #endif
33 };
34 int Derived::sNum = 20;
35 void test01()
36 {
  Derived p1;
37
  //如果子类和父类有同名的静态成员变量,父类中的静态成员变量会被隐藏
  cout << p1.sNum << endl;</pre>
39
  //如果子类和父类有同名的静态成员函数,父类中的静态成员函数都会被隐藏
40
  p1.getNum(1,2);
41
42
43 }
44
45 int main()
46 {
  test01();
47
48
   return 0;
49
  }
50
51
```

7 多继承

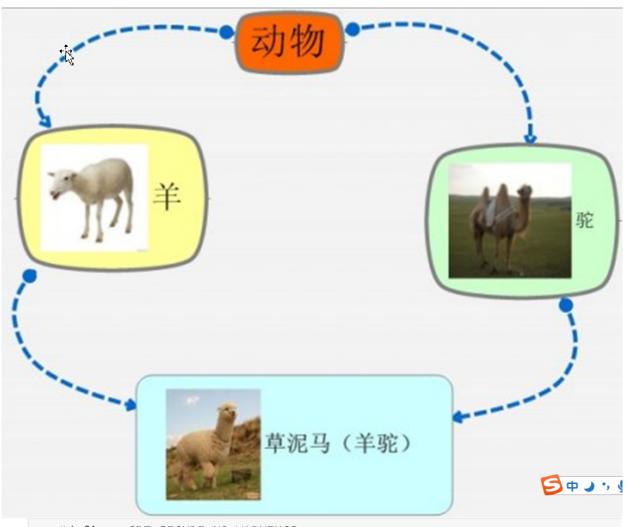
7.1 多继承的概念

一个类继承了多个类

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
```

```
8 class A
9 {
10 public:
11 int a;
12
13 };
14
15 class B
16 {
17 public:
18 int a;
19
20 };
21
22 class C :public A, public B
23 {
24 public:
25 int c;
26
27 };
28 void test01()
30
31 C p;
32 p.A::a = 10;
33 p.B::a = 10;
34 //p.b = 20;
35 p.c = 30;
36 }
37 int main()
38 {
39 test01();
40 return 0;
41 }
```

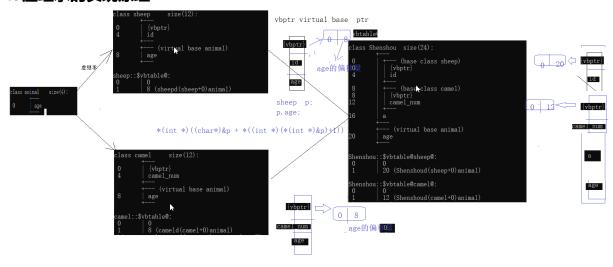
7.2菱形继承和虚继承



```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 class animal
8 {
9 public:
10 int age;
11 };
12 class sheep:virtual public animal
13 {
14 public:
15 int id;
16 };
17 class camel:virtual public animal
18 {
19 public:
```

```
20
    int camel_num;
21
22
   };
  class Shenshou:public sheep,public camel
23
24
  public:
25
    int a;
26
27 };
   void test01()
   {
29
30
    Shenshou p;
31
   //p.sheep::age = 100;
32
   p.age = 100;
33
34 }
   int main()
35
36
37
   return;
38
39
```

7.3虚继承的实现原理



八.多态

1 多态的概念

多态: 一种接口,多种形态

静态多态:编译时,地址早绑定(静态联编)foo(int)foo()

动态多态: 运行时,才确定需要调用的地址(动态联编)

发送多态的四个条件:

- 父类中有虚函数
- 必须发送继承
- 子类必须重写虚函数(函数的返回值 函数名 参数一致 函数内容可以不一致)
- 父类的指针或引用指向子类的对象

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 class Animal
8 {
9 public:
10 virtual void speak() //虚函数
11 {
12 cout << "动物在说话" << endl;
13 }
14
15 };
16
17 class Dog:public Animal
18 {
19 public:
20 //重写虚函数 函数的返回值 参数 函数名一致
21 void speak()
22 {
23 cout << "狗在说话" << endl;
24 }
25
26 };
27
28 class Cat :public Animal
29 {
30 public:
31 void speak()
32 {
33 cout << "猫在说话" << endl;
```

```
34
35
  };
36
37 //如果两个类发生了继承 父类和子类编译器会自动转换.不需要人为转换
  void do_work(Animal &obj)
39
  {
   obj.speak();//地址早绑定 -> 加上函数前面加上virtual关键字 地址晚绑定
  }
41
42
43 void test01()
44 {
45
   Animal p1;
   do_work(p1);
46
47
48
   Dog p2;
   do_work(p2);
49
50
  Cat p3;
51
   do_work(p3);
53 }
  int main()
  {
55
56
  test01();
  return 0;
58 }
```

2 多态实现计算器的案例

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;

//开发时 对源码的修改是关闭的 对扩展是开发的
class Mycalc

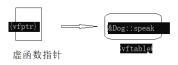
public:
    int calc(int a, int b, string cmd)
```

```
if (cmd == "+")
   {
15
   return a + b;
16
17
   else if (cmd == "-")
18
19
20
   return a - b;
21
   else if (cmd == "*")
22
23
  return a * b;
24
  }
25
  }
26
27 };
28 void test01()
29 {
30 Mycalc p;
31 cout << p.calc(3, 4, "+") << endl;</pre>
  cout << p.calc(3, 4, "-") << endl;</pre>
32
33 }
35 //多态实现计算器案例
36 class Calc
37 {
38 public:
  virtual int mycalc(int a, int b)
40 {
41 return 0;
42
  }
43 };
44
45 class Add :public Calc
46 {
47 public:
48
  int mycalc(int a, int b)
49
  {
50
  return a + b;
51
   }
53 };
```

```
54 class Sub :public Calc
55 {
56 public:
    int mycalc(int a, int b)
58
   return a - b;
60
61 };
62 class Mul :public Calc
63 {
64 public:
    int mycalc(int a, int b)
65
66
   return a * b;
67
   }
68
69 };
70
71 int do_calc(int a, int b, Calc &obj)
72
   {
73
   return obj.mycalc(a, b);
74
75 }
76
77 void test02()
78 {
   Add p;
   cout << do_calc(2,3,p) <<endl;</pre>
80
81
    Sub p1;
   cout << do_calc(2, 3, p1) << endl;</pre>
82
  Mul p2;
83
   cout << do_calc(2, 3, p2) << endl;</pre>
85 }
86
87 int main()
88
  {
   test02();
89
90
91 }
```

```
class Animal {
  public:
    virtual void speak() //虚函数
    {
        cout << "动物在说话" << endl;
    }
};
```

Dog p;//当定义Dog类型对象时会先构造父类的对象



p. speak();

4 纯虚函数和抽象类

纯虚函数: 将虚函数 等于0 实质是将虚函数 表的函数入口地址置为NULL 抽象类:一个类中如果有纯虚函数,那么这个类就是一个抽象类,抽象类不能实例化对象继承抽象类的子类也是一个抽象类,如果子类重写了虚函数,那么子类就不是抽象类

```
2 //多态实现计算器案例
3 class Calc
4 {
5 public:
 virtual int mycalc(int a, int b) = 0;//虚函数等于0 纯虚函数
 /*{
 return 0;
  }*/
10 };
11 class Mod:public Calc
12 {
13 public:
  //子类继承了抽象类,那么子类也是一个抽象类
14
  int mycalc(int a, int b){}//如果子类重写类虚函数 就不是抽象类
16 };
17
  //如果有纯虚函数的类 叫做抽象类 抽象类不能实例化对象
19 void test03()
20 {
   //Calc p;
   Mod p1;
23
```

5 纯虚函数和多继承

多继承带来了一些争议,但是接口继承可以说一种毫无争议的运用了。

绝大数面向对象语言都不支持多继承,但是绝大数面向对象对象语言都支持接口的概念, **C++**中没有接口的概念,但是可以通过纯虚函数实现接口。

接口类中只有函数原型定义,没有任何数据定义。

多重继承接口不会带来二义性和复杂性问题。接口类只是一个功能声明,并不是功能实现, 子类需要根据功能说明定义功能实现。

注意:除了析构函数外,其他声明都是纯虚函数。

6 虚析构

作用:在调用基类的析构函数之前,会先调用子类的析构函

```
1 #define CRT SECURE NO WARNINGS
2 #include <iostream>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <string>
6 using namespace std;
7 class Animal
8 {
9 public:
  virtual void speak() //虚函数
11 {
12 cout << "动物在说话" << endl;
13
   virtual ~Animal()//虚析构 作用 在调用基类的析构函数之前,会先调用子类的析构函
14
数
16 cout << "Animal的析构" << endl;
17 }
18 };
19
20 class Dog :public Animal
21 {
22 public:
23 //重写虚函数 函数的返回值 参数 函数名一直
24 void speak()
```

```
25 {
  cout << "狗在说话" << endl;
27
  ~Dog()
28
  {
29
  cout << "狗的析构" << endl;
31
32
33 };
34
35 void do_work(Animal &obj)
36 {
  obj.speak();//地址早绑定 -> 加上函数前面加上virtual关键字 地址晚绑定
37
38 }
39 void test01()
40 {
41 Animal *p = new Dog;
42 p->speak();
43 delete p;
44
45 }
46 int main()
47 {
48 test01();
49 return 0;
50 }
```

7 纯虚析构

虚析构函数等于0

```
1 class Animal
2 {
3 public:
4 virtual void speak() //虚函数
5 {
6 cout << "动物在说话" << endl;
7 }
8 virtual ~Animal() = 0;//纯虚析构
9 /*{
```

```
10 cout << "Animal的析构" << endl;
11 }*/
12 };
```

8 重载 重定义 重写

重载:

- 函数名相同
- 同一个作用域
- 参数的个数,顺序,类型不一致
- const也可以成为重载的条件

重定义:

- 发生继承
- 子类和父类有同名的变量和函数,父类中同名的变量和函数会被隐藏

重写:

- 父类中有虚函数
- 发生了继承
- 子类重写了虚函数 函数名 返回值 参数一致,函数体不一致