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];

};
```

```cpp
//c++中 可以放函数
struct _stu1
{
 int a;
 int b[5];
 void print_stu()
 {
 cout << a << endl;
 }

};

struct student
{
 //学生的属性和数据
 int age;
 int id;
 char name[256];
 //操作属性的叫做 方法或行为-函数
 void print()
 {
 cout << age << id << name << endl;
 }


};
void test01()
{
 student obj;
 obj.age = 10;
 obj.id = 20;
 strcpy(obj.name,"lucy");
 obj.print();

}
int main()
{
 test01();
 return 0;
}
```

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

```c
#include <stdio.h>
//表示人
struct Person
{
 int age;
 char name[128];
};
void Person_eat(struct Person *p)
{
 printf("%s 在吃饭\n",p->name);
}

//表示dog
struct Dog
{
 int age;
 char name[128];
};
void Dog_eat(struct Dog *p)
{
 printf("%s 在吃粑粑\n", p->name);
}


void test01()
{
 struct Person p1;
 p1.age = 20;
 strcpy(p1.name,"bob");
 Person_eat(&p1);

 struct Dog d1;
 d1.age = 7;
 strcpy(d1.name,"旺财");
 Dog_eat(&d1);

```

```
 Dog_eat(&p1);//人 调用了狗的行为


}
int main()
{
 test01();
 return 0;
}
```

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

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
using namespace std;
//c++中对事物的封装 将属性和行为封装在一起
//类 将事物抽象成属性和行为,并且封装在一起
//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
// struct Person
class Person
{
public://公有的
 //类中的所有成员 访问的权限都是私有的 private
 //属性
 int age;
 char name[128];
 //行为
 void Person_eat()
 {
 printf("%s 吃饭\n",name);
 }
};
```

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

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
using namespace std;
class person
{
public: //公有的 类内类外都可访问
 int mTall; //多高，可以让外人知道
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

 void show()
 {
 cout << mTall << " ";
 cout << mMoney << " ";
 cout << mAge << " ";
 }

};
void test01()
{
 person p;
 p.mTall = 180;


}

int main()
{

 return 0;
}
```

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

```
class AccessLevels{
public:
//对只读属性进行只读访问
int getReadOnly(){ return readOnly; }
//对读写属性进行读写访问
void setReadWrite(int val){ readWrite = val; }
int getReadWrite(){ return readWrite; }
//对只写属性进行只写访问
void setWriteOnly(int val){ writeOnly = val; }
 private:
 int readOnly; //对外只读访问
 int noAccess; //外部不可访问
 int readWrite; //读写访问
 int writeOnly; //只写访问
 };
```

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()
```

```
15  {
16  cout << m_name << " " << m_age << endl;
17  }
18  int get_age()
19  {
20  return m_age;
21  }
22  void set_age(int age)
23  {
24  if (age >= 0 && age <= 100)
25  {
26  m_age = age;
27  }
28  }
29  char *get_name()
30  {
31  return m_name;
32  }
33  void set_name(char *name)
34  {
35  strcpy(m_name,name);
36  }
37
38 private:
39  int m_age;
40  char m_name[128];
41
42 };
43
44 void test01()
45 {
46  Person p1;
47  p1.person_init(20,"lucy");
48  p1.show_person();
49  p1.set_age(30);
50  p1.set_name("bob");
51  p1.show_person();
52 }
53 int main()
54 {
```

```
 test01();
 return 0;
}
```

二. 面向对象程序设计案例

2.1 立方体案例

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
using namespace std;
class Cube
{
public:
 void set_L(int l)
 {
 L = l;
 }
 void set_W(int w)
 {
 W = w;
 }
 void set_H(int h)
 {
 H = h;
 }
 int get_L()
 {
 return L;
 }
 int get_W()
 {
 return W;
 }
 int get_H()
 {
 return H;
 }
 //求立方体的体积
 int get_cube_V()
```

```
 {
 return L*W*H;
 }
 //求立方体面积
 int get_cube_S()
 {
 return 2 * W*L + 2*W*H + 2 * L*H;
 }
 //判断两个立方体是否相等
 bool compare_cube(Cube &c1)
 {
 return c1.get_L() == L && c1.get_W() == W && c1.get_H() == H;
 }

private:
 int L;
 int W;
 int H;
};

bool comapre_cube(Cube &c1, Cube &c2)
{

 return c1.get_L() == c2.get_L()&& c1.get_W() == c2.get_W() &&
c1.get_H() == c2.get_H();
}

void test01()
{
 Cube c1;
 c1.set_L(10);
 c1.set_W(20);
 c1.set_H(30);
 cout << c1.get_cube_S() << endl;
 cout << c1.get_cube_V() << endl;

 Cube c2;
 c2.set_L(20);
 c2.set_W(20);
 c2.set_H(30);

```

```
 if (c1.compare_cube(c2))
 {

 cout << "立方体相等" << endl;
 }
 else
 {
 cout << "立方体不相等" << endl;
 }

 if (comapre_cube(c1, c2))
 {
 cout << "立方体相等" << endl;
 }
 else
 {
 cout << "立方体不相等" << endl;
 }



}
int main()
{
 test01();
 return 0;
}
```

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; }
 private:
 int mX;
 int mY;
};

//圆类
class Circle {
public:
 void setP(int x, int y) {
 mP.setX(x);
 mP.setY(y);
 }
 void setR(int r) { mR = r; }
 Point& getP() { return mP; }
 int getR() { return mR; }
 //判断点和圆的关系
 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) {
 cout << "Point(" << point.getX() << "," << point.getY() << ")在圆内!" <<
endl;
 }
 else if (distance > radius) {
 cout << "Point(" << point.getX() << "," << point.getY() << ")在圆外!" <<
endl;
 }
 else {
 cout << "Point(" << point.getX() << "," << point.getY() << ")在圆上!" <<
endl;
 }
 }
private:
 Point mP; //圆心
 int mR; //半径
};

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

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

class Person
{
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 可以发生函数重载
  Person(int age,string name)
  {
  cout << "person的构造函数" << endl;
  m_age = age;
  m_name = name;
  }
  //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
  ~Person()
  {
  cout << "析构函数" << endl;
  }
  int m_age;
  string m_name;
};

void test01()
{
  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
}
```

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

 return 0;
}
```

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:

```
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

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <string>
using namespace std;
class Person
{
public:
 //有参和无参构造
 Person()
 {
 cout << "无参构造" << endl;
 }

 Person(int a, string n)
 {
 cout << "有参构造" << endl;
```

```
 age = a;
 name = n;
 }
 // 拷贝构造的调用时机 : 旧对象初始化新对象
 //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
 Person(const Person &p)// Person p = p2
 {

 //拷贝构造做了简单的值拷贝
 age = p.age;
 name = p.name;
 cout << "拷贝构造" << endl;

 }
 int age;
 string name;

};
void test01()
{
 //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
 Person p1;//调用无参构造时 不能使用括号法
 Person p2(10,"lucy");
 Person p3(p2);//调用系统提供的默认拷贝构造

}
int main()
{
 test01();
 return 0;
}
```

//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

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

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

```
 Person (10, "lucy");// 调用了有参构造创建了一个匿名对象
 Person ();//调用了无参构造创建了一个匿名对象
 Person p1(20,"heihei");
 //Person (p1);//在定义时匿名对象不能使用括号法调用拷贝构造
}

//显示法调用构造函数

void test03()
{
 Person p1 = Person(10, "lucy");//显示法调用有参构造
 Person p2 = Person(p1);//显示法调用拷贝构造
 Person p3 = Person();//显示法调用无参构造

}

// 隐式法调用构造函数
void test04()
{
 Person p1 = { 10, "lucy" };// 隐式法调用有参构造
 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

}
```

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 {
```

```
public:
 Person() {
 cout << "no param contructor!" << endl;
 mAge = 10;
 }
 Person(int age) {
 cout << "param constructor!" << endl;
 mAge = age;
 }
 Person(const Person& person) {
 cout << "copy constructor!" << endl;
 mAge = person.mAge;
 }
 ~Person() {
 cout << "destructor!" << endl;
 }
public:
 int mAge;
};
//1. 旧对象初始化新对象
void test01() {

 Person p(10);
 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
}

//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

void test02() {
 Person p(10);
 doBussiness(p);
}

//3. 函数返回局部对象
Person MyBusiness() {
 Person p(10);
 cout << "局部p:" << (int*)&p << endl;
```

```
 return p;
}
void test03() {
 //vs release、qt下没有调用拷贝构造函数
 //vs debug下调用一次拷贝构造函数
 Person p = MyBusiness();
 cout << "局部p:" << (int*)&p << endl;
}
int main()
{
 test03();
 return 0;
}

```

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)

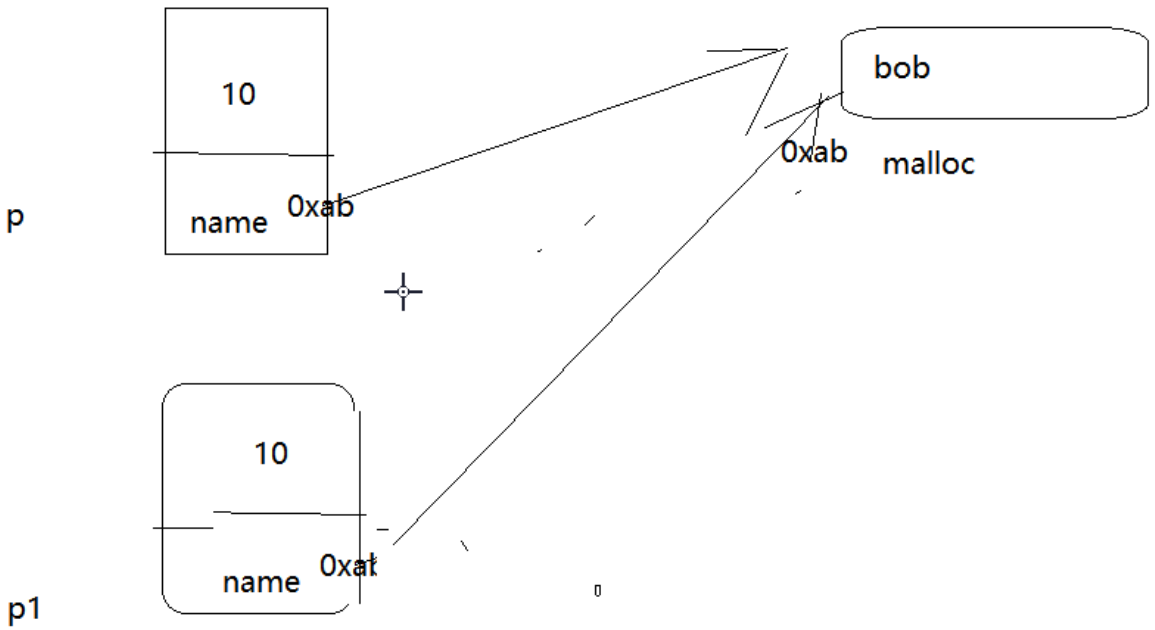
3．默认拷贝构造函数，对类中非静态成员属性简单值拷贝

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

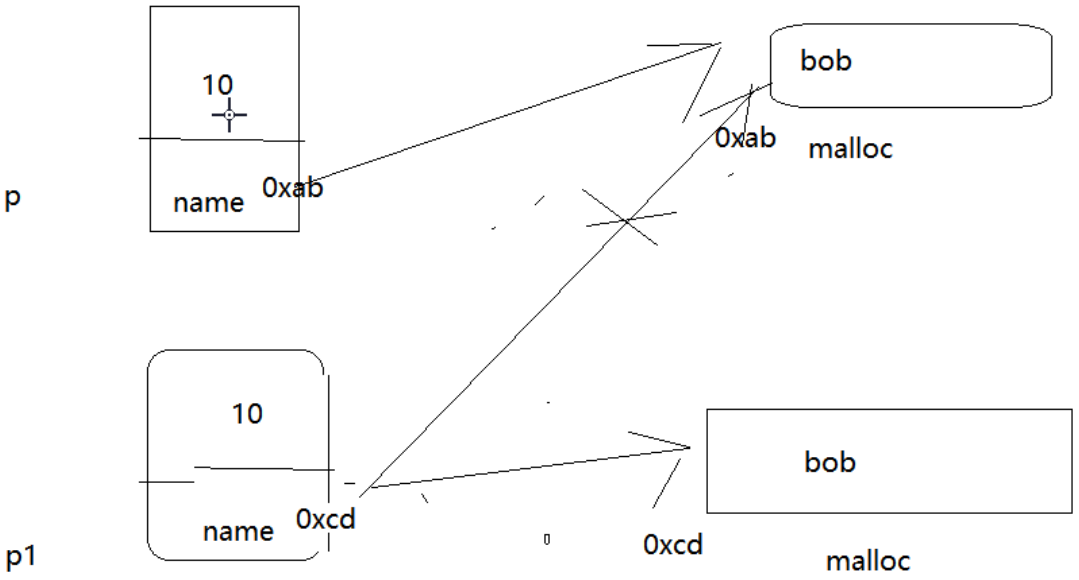
3.6.1 浅拷贝




```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <string>
using namespace std;
class Person
{
public:
 Person(int age,char *str)
  {
  cout << "有参构造" << endl;
  mAge = age;
  name = (char *)malloc(strlen(str)+1);
  strcpy(name, str);
  }
  void show()
  {
  cout << name << " " << mAge << endl;
  }
  ~Person()
  {
  if (name != NULL)
  {
  free(name);
  name = NULL;
```

```
26   }
27   cout << "析构" << endl;
28   }
29
30   int mAge;
31   char *name;
32
33
34  };
35  void test01()
36  {
37   Person p(10,"bob");
38   p.show();
39   Person p1(p);//调用拷贝构造 函数
40   p1.show();
41
42  }
43  int main()
44  {
45   test01();
46
47   return 0;
48  }
```

3.6.2 深拷贝




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

```
{
mAge = p.mAge;
name = (char *)malloc(strlen(p.name)+1);
strcpy(name,p.name);

}
```

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

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <string>
using namespace std;
class person
{
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)
 {
 m_a = a;
 m_b = b;
 m_c = c;
 }*/
 //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
 //调用构造函数是 定义并初始化,顺序和声明的顺序一致
 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

 void show()
 {
 cout << m_a << " " << m_b << " " << m_c << endl;
 }
 int m_a;
```

```
 int m_c;
 int m_b;
};
void test01()
{

 person p1(2,3,5);
 p1.show();
}
int main()
{
 test01();
 return 0;
}
```

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

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <string>
using namespace std;
class Phone
{
public:
 Phone(string name)
 {
 cout << "Phone的构造函数" << endl;
 pho_name = name;
 }
 ~Phone()
 {
 cout << "Phone的析构" << endl;
 }
 string pho_name;
```

```
19
20 };
21 class Game
22 {
23 public:
24  Game(string name)
25  {
26  cout << "Game的构造函数" << endl;
27  game_name = name;
28  }
29  ~Game()
30  {
31  cout << "Game的析构" << endl;
32  }
33  string game_name;
34 };
35 class Person
36 {
37 public:
38  /*Person(string per_name1,string pho_name,string g_name)
39  {
40  per_name = per_name1;
41  phone.pho_name = pho_name;
42  game.game_name = g_name;
43  }*/
44  Person(string per_name1, string pho_name, string g_name) :per_name(per_
name1), phone(pho_name), game(g_name) {
45  cout << "person的构造函数" << endl;
46  }
47  void show()
48  {
49  cout << per_name << phone.pho_name << " 玩着" << game.game_name << endl;
50  }
51  ~Person()
52  {
53  cout << "Person的析构" << endl;
54  }
55  string per_name;
56  Game game;
57  Phone phone;
```

```
58
59 };
60 void test01()
61 {
62
63  Person p1("bob","诺基亚","贪吃蛇");
64  p1.show();
65
66 }
67
68 int main()
69 {
70  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

```
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
8  class person
9  {
10  public:
11   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()
28  {
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;

```
1  #define _CRT_SECURE_NO_WARNINGS
2  #include <iostream>
3  #include <string.h>
```

```cpp
#include <stdlib.h>
#include <string>
using namespace std;
class person
{
public:
 person()
 {
 cout << "无参构造" << endl;
 }
 ~person()
 {
 cout << "析构" << endl;
 }

 int age;

};
void test01()
{
 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
 cout << *p << endl;
 *p = 100;
 cout << *p << endl;
 delete p;//释放申请的空间

}


//申请一个对象
void test02()
{
 person *p = new person;//sizeof(person)
 delete p;
}

//申请一个数组
void test03()
{
 //new一个数组时,返回的是该数组的首元素的地址
```

```
 int *p = new int[10];
 for (int i = 0; i < 10; i++)
 {
 p[i] = i + 100;
 }
 for (int i = 0; i < 10; i++)
 {
 cout << p[i] << " ";
 }
 cout << endl;
 delete []p;

}
int main()
{
 test03();
 return 0;
}
```

3.9.3 用于数组的new和delete

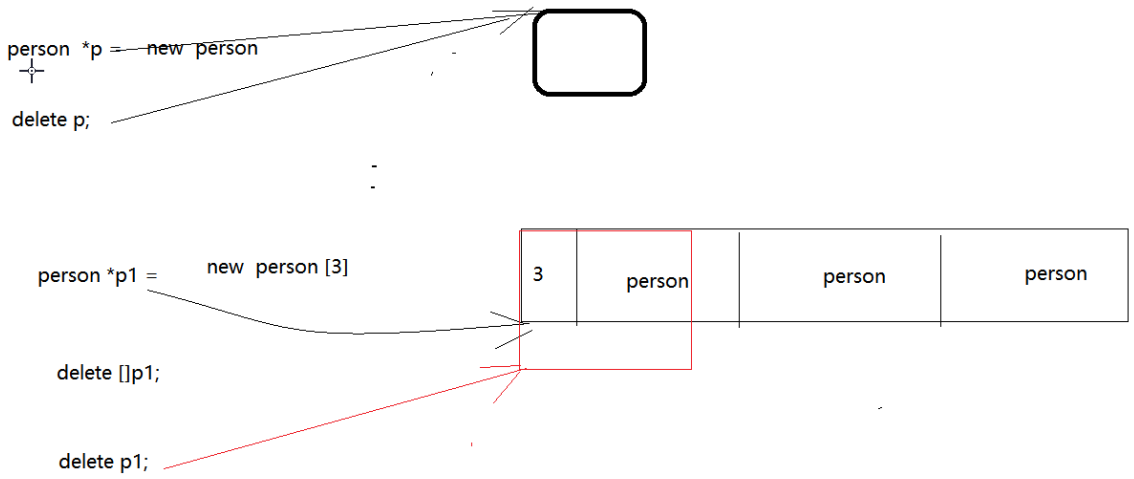
```
void test04()
{
 //new时调用有参构造
 person *p = new person(10);
 delete p;
 person *p1 = new person[10];//注意new对象的数组时不能调用有参构造 只能调用无参构造
 delete []p1;

}
```

3.9.4 delete void *问题

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

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

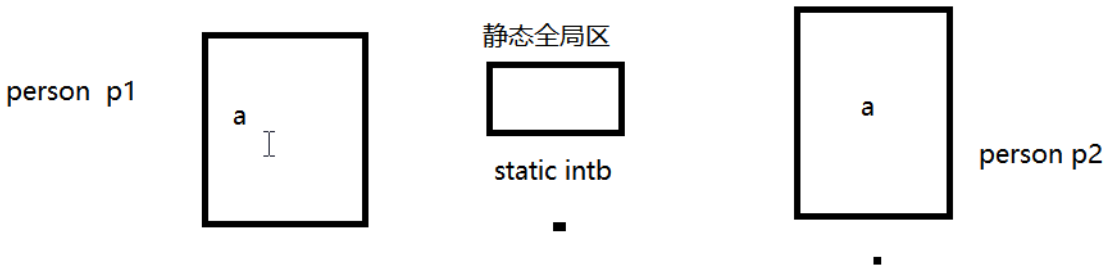


3.10 静态成员

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>
```

```cpp
#include <string>
using namespace std;
class person
{
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
  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
};
int person::b = 10;//类中成员变量的定义
void test01()
{
 person p1;
 p1.b = 100;
 cout << p1.b << endl;

}
void test02()
{
 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;

}

using namespace std;
int main()
{
 test02();

 return 0;
}
```

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

```cpp
#define _CRT_SECURE_NO_WARNINGS
```

```cpp
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class person
{
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
  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()
  {
  cout << a << " " << b << endl;;
  }
  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
  {
  cout << " " << b << endl;;
  }

};
int person::b = 100;
void test01()
{
 person::static_show();//通过流类的作用域访问静态成员函数
 person p1;
 p1.static_show();//通过对象访问静态成员函数

}
int main()
{
 test01();
 return 0;
}
```

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

```
const int num = 10;//const修饰的全局变量保存在常量区 不可更改
class person
{
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
 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
 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
};
int person::b = 10;//类中成员变量的定义

void test()
{
 cout << person::c << endl;
 person p1;
 cout << p1.c << endl;

}
```

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

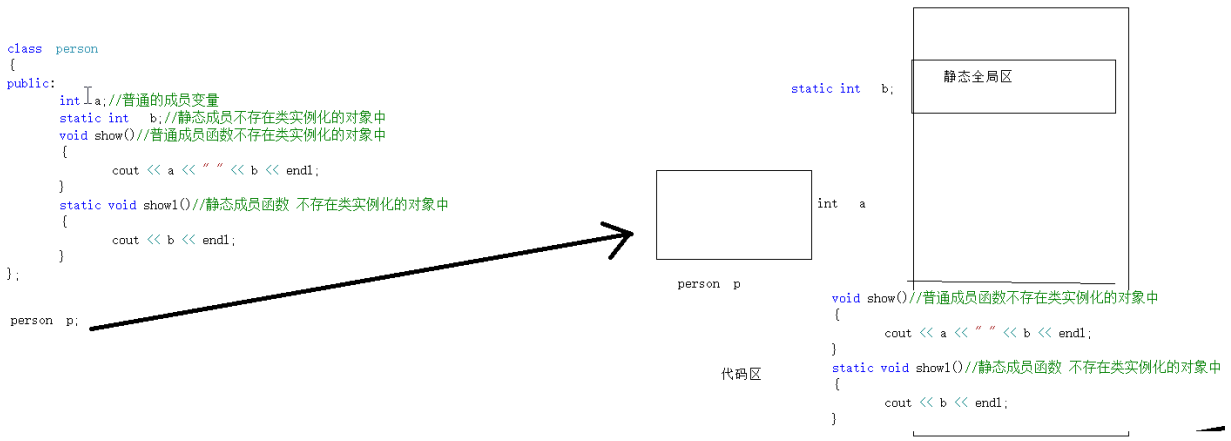
```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class Feifei
{
public:
  int age;
  int yanzhi;
  static Feifei * instance()
  {
  return single;
  }
 private:
  static Feifei *single;
  Feifei() //无参构造私有化
  {}
  Feifei(const Feifei &p)
  {}


 };
 Feifei * Feifei::single = new Feifei;

 void test03()
 {
  Feifei * p = Feifei::instance();
  p->age = 10;
  p->yanzhi = 20;

  Feifei * p1 = Feifei::instance();
  cout << p1->age << " " << p1->yanzhi << endl;

```

```
36
37
38  }
39  void test02()
40  {
41   //Feifei::single->age = 100;
42   //Feifei::single->yanzhi = 100;
43
44   //Feifei p1(*Feifei::single);//调用拷贝构造实例化出一个对象
45   // Feifei::single = NULL;
46
47  }
48  void test01()
49  {
50   //Feifei p1;//需要调用无参构造
51   //Feifei p2;
52
53  }
54  int main()
55  {
56   test03();
57   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

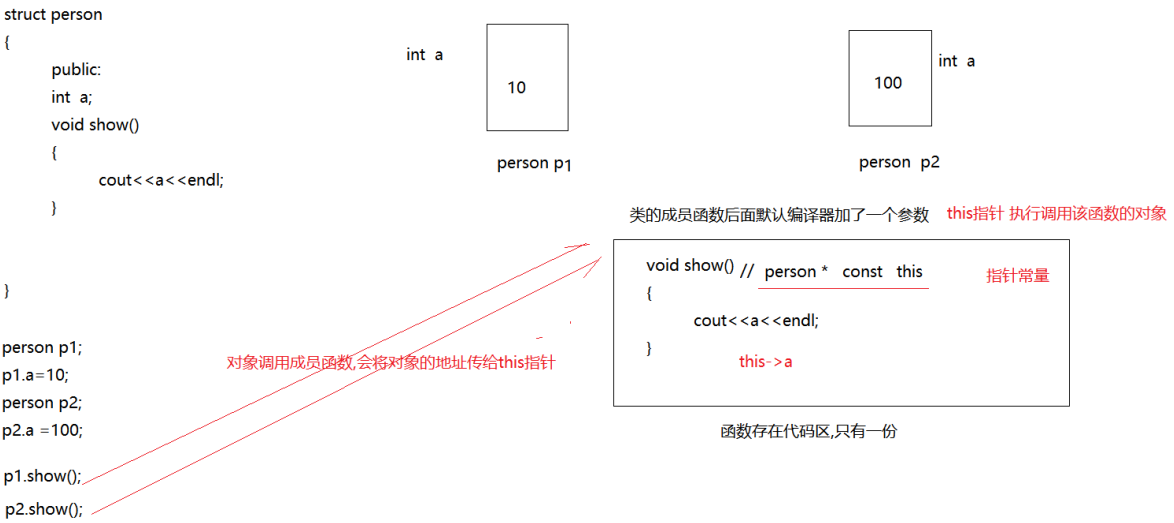



```cpp
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class person
{
public:
 int a;//普通的成员变量
 static int b;//静态成员不存在类实例化的对象中
 void show()//普通成员函数不存在类实例化的对象中
 {
 cout << a << " " << b << endl;
 }
 static void show1()//静态成员函数 不存在类实例化的对象中
 {
 cout << b << endl;
 }
};
int person::b = 1;
void test01()
{
 person p;
 p.show();
 //空类的大小不是0 而是1
 cout << sizeof(person) << endl;

}
int main()
```

```
{
 test01();

}
```

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

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class person
{
public:
 person(int age, string name)// this
 {
 this->age = age;
 this->name = name;
 }
 void show()
```

```
 {
 cout << age << " " << name << endl;
 }
 person person_add( person &p2)//this ------> p1
 {
 person p(this->age+p2.age,this->name+p2.name);//"helloworld"
 return p;
 }

 int age;
 string name;

};
person person_add(person &p1, person &p2)
{
 person p(p1.age+p2.age,p1.name+p2.name);//"helloworld"
 return p;
}
void test02()
{
 person p1(10, "hello");
 person p2(20, "world");
 //p3 = p1 + p2 30,"helloworld"
 //person p3 = person_add(p1,p2);
 //p3.show();
 person p3 = p1.person_add(p2);
 p3.show();



}
void test01()
{
 person p1(10,"lucy");
 p1.show();

}
int main()
{
 test02();

```

```
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

```cpp
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class Building
{
 friend void print_Building(Building &b);
public:
 Building(string hall,string bedroom)
 {
 this->bedroom = bedroom;
 this->hall = hall;
 }
 string hall;
private:
 string bedroom;

};

void print_Building(Building &b)
{
 cout << b.hall << " " << b.bedroom << endl;
}

void test01()
{
 Building b1("凌霄殿","闺房");
 print_Building(b1);
}
int main()
{
 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

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

```cpp
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;

class Building;
class Goodgay
{
public:
 Goodgay(string hall, string bedroom);

 void visit();
 Building *b;
};
class Building
{
 //friend void print_Building(Building &b);
 //friend class Goodgay; //一个类成为另一个类的友元
 friend void Goodgay::visit();//类的成员函数成为另一类的友元
public:
 Building(string hall, string bedroom)
 {
 this->bedroom = bedroom;
 this->hall = hall;
 }
 string hall;
private:
 string bedroom;

};

Goodgay::Goodgay(string hall, string bedroom)
{
 b = new Building( hall, bedroom);
}

void Goodgay::visit()
{
 cout << b->hall << " " << b->bedroom << endl;
}
```

```
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;'

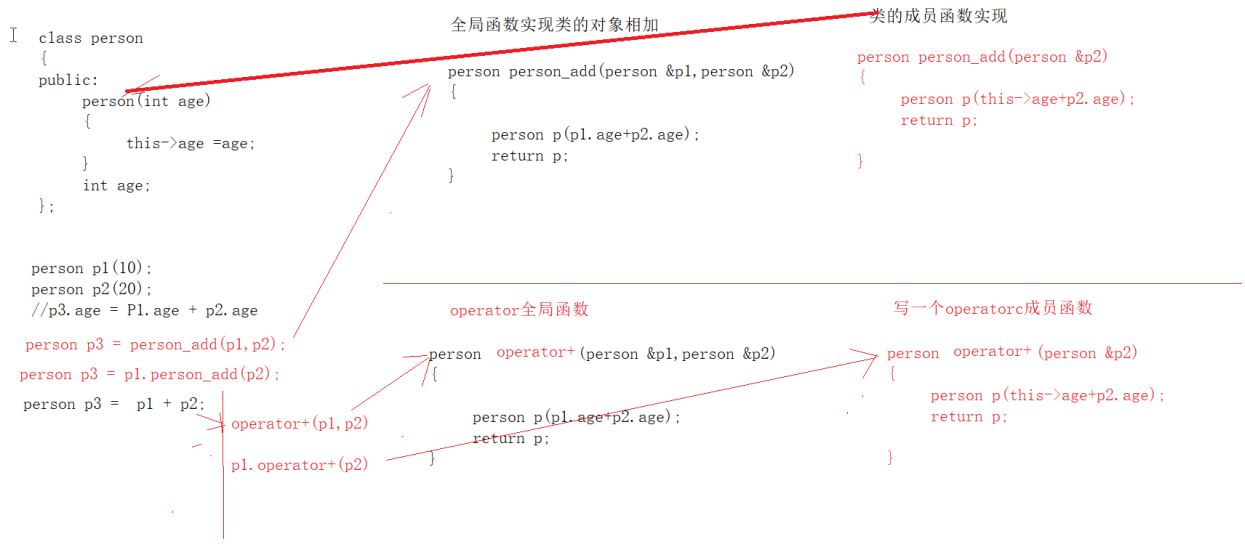
int c = a +b;

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



2 重载加号运算符

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class person
{
public:
  person(int age)
  {
  this->age = age;
  }

  person operator+( person &p2)
  {
  person p(this->age+p2.age);
  return p;
  }

  int age;

};

//person operator+(person &p1, person &p2)
//{
```

```
// person p(p1.age+p2.age);
// return p;
//}

void test01()
{
 person p1(10);
 person p2(20);
 person p3 = p1 + p2;// operator+(p1,p2) p1.operator+(p2)
 cout << p3.age << endl;
}


int main()
{
 test01();
 return 0;
}

```

3 重载左移运算符和算符重载碰上友元函数

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class person
{
 friend ostream& operator<<(ostream &cout, person &p);
public:
 person(int age)
 {
 this->age = age;
 }
private:
 int age;

};
```

```cpp
ostream& operator<<(ostream &cout, person &p)
{
 cout << p.age;
 return cout;
}
void test01()
{
 person p1(10);
 cout << p1 <<endl;
 // operator<<(cout,p1) //cout.operator<<(p1)

}
int main()
{
 test01();

 return 0;
}
```

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

+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%
^=	&=	|=	<<	>>	>>=	<<=	==	!=
<=	>=	&&	||	++	--	->*	‘	->
[]	()	new	delete	new[]	delete[]			

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

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class Myint
{
public:
  Myint(int num)//this
  {
  this->num = num;
  }
  Myint& operator++()
  {
```

```
 this->num = this->num + 1;
 return *this;
 }
 Myint operator++(int)
 {
 Myint tmp = *this;
 //加加
 this->num = this->num + 1;
 return tmp;

 }
 int num;
};
ostream & operator<<(ostream &cout, Myint &p)
{
 cout << p.num;
 return cout;

}
void test01()
{
 Myint p1(10);
 cout << p1 << endl;
 ++p1;//operator++(p1) p1.operator++()
 cout << ++p1 << endl;
 cout << p1++ << endl;//p1.operator++(int)
}
int main()
{
 test01();

 return 0;
}
```

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

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class person
{
public:
 person(int age)
 {
 this->age = age;
 }
 int age;
};
class Smartpointer
{
public:
 Smartpointer(person *p1)
 {
 this->p = p1;
 }
 ~Smartpointer()
 {
 delete p;
 cout << "释放了p" << endl;
 }
 person *operator->()
 {
 return p;
 }
 person& operator*()
 {
 return *p;
 }
 person *p;
};
```

```
void test01()
{
 // 局部对象 在释放之前可以帮助释放 p,
 //person *p = new person(10);
 Smartpointer sp(new person(10));
 //cout << p->age << endl;
 cout << sp->age << endl;//sp-> 返回的是p p sp.operator->()
 cout << (*sp).age << endl;//sp.operator*()
 //忘记释放p指向申请的对象
}
int main()
{
 test01();
 return 0;
}
```

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=()

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class person
{
public:
  person() {}
  person(int age1,char *name1)
  {
  age = age1;
  name = new char[strlen(name1) + 1];
  strcpy(name,name1);
```

```
 }
 person& operator=(person &p1)//this- ..> p2
 {
 this->age = p1.age;
 this->name = new char[strlen(p1.name)+1];
 strcpy(this->name, p1.name);

 return *this;
 }
 ~person()
 {
 delete []name;
 }
 int age;
 char *name;

};
void test01()
{
 person p1(10,"bob");
 person p2 ;
 p2 = p1;//p2.operator=(p1)
 cout << p2.age << " " << p2.name << endl;

}
int main()
{
 test01();
 return 0;
}
```

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
{
```

```
public:
 person(int age,string name)
 {
 this->age = age;
 this->name = name;
 }
 bool operator==(person &p2)//this->>>p1
 {
 return this->age == p2.age && this->name == p2.name;
 }
 bool operator!=(person &p2)//this->>>p1
 {
 return this->age != p2.age || this->name != p2.name;
 }
 int age;
 string name;
};
void test01()
{
 person p1(10, "lucy");
 person p2(20, "lucy");
 if (p1 == p2)//p1.operator==(p2)
 {
 cout << "p1 == p2" << endl;
 }
 if (p1 != p2)//p1.operator!=(p2)
 {
 cout << "p1 != p2" << endl;
 }


}
int main()
{
 test01();
 return 0;
}
```

9 函数对象

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

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class Myadd
{
public:

  int add(int a, int b)
  {
  return a + b;
  }
  int operator()(int x, int y)
  {
  return x+ y;
  }
};

void test()
{
  //Myadd p;
  //cout << p.add(3, 4) << endl;
  // p() 可以像函数一样调用的对象 函数对象
  //cout << p(3, 4) << endl;//p.operator()(3,4)
  cout << Myadd ()(3, 4) << endl;//匿名对象 Myadd ().operator()(3,4)

}
int main()
{
  test();

  return 0;
}
```

10 不要重载&&和||

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。

```
class Complex{
public:
Complex(int flag){
this->flag = flag;
}
Complex& operator+=(Complex& complex){
this->flag = this->flag + complex.flag;
return *this;
}
 bool operator&&(Complex& complex){
 return this->flag && complex.flag;
 }
 public:
 int flag;
 };
 int main(){

 Complex complex1(0);  //flag 0
 Complex complex2(1);  //flag 1

 //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
 //这边却是，先运算（complex1+complex2），导致，complex1的flag变为complex1+complex2的值， complex1.a = 1
 // 1 && 1
 //complex1.operator&&(complex1.operator+=(complex2))
 if (complex1 && (complex1 += complex2)){
         //complex1.operator+=(complex2)
 cout << "真!" << endl;
 }
 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;
6
7 class MyString
8 {
9 friend ostream& operator<< (ostream  & out, MyString& str);
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
```

```
//=号重载
MyString& operator=(const char * str);
MyString& operator=(const MyString& str);

//字符串拼接 重载+号
MyString operator+(const char * str );
MyString operator+(const MyString& str);

//字符串比较
bool operator== (const char * str);
bool operator== (const MyString& str);
private:
char * pString; //指向堆区空间
int m_Size; //字符串长度 不算'\0'
};
MyString.cpp
#include "MyString.h"

//左移运算符
ostream& operator<< (ostream & out, MyString& str)
{
out << str.pString;
return out;
}
//右移运算符
istream& operator>>(istream& in, MyString& str)
{
//先将原有的数据释放
if (str.pString != NULL)
{
delete[] str.pString;
str.pString = NULL;
}
char buf[1024]; //开辟临时的字符数组，保存用户输入内容
in >> buf;

str.pString = new char[strlen(buf) + 1];
strcpy(str.pString, buf);
str.m_Size = strlen(buf);

```

```
return in;
}

//构造函数
MyString::MyString(const char * str)
{
this->pString = new char[strlen(str) + 1];
strcpy(this->pString, str);
this->m_Size = strlen(str);
}

//拷贝构造
MyString::MyString(const MyString& str)
{
this->pString = new char[strlen(str.pString) + 1];
strcpy(this->pString, str.pString);
this->m_Size = str.m_Size;
}
//析构函数
MyString::~MyString()
{
if (this->pString!=NULL)
{
delete[]this->pString;
this->pString = NULL;
}
}

char& MyString::operator[](int index)
{
return this->pString[index];
}

MyString& MyString::operator=(const char * str)
{
if (this->pString != NULL){
delete[] this->pString;
this->pString = NULL;
}
this->pString = new char[strlen(str) + 1];
```

```
strcpy(this->pString, str);
this->m_Size = strlen(str);
return *this;
}

MyString& MyString::operator=(const MyString& str)
{
if (this->pString != NULL){
delete[] this->pString;
this->pString = NULL;
}
this->pString = new char[strlen(str.pString) + 1];
strcpy(this->pString, str.pString);
this->m_Size = str.m_Size;
return *this;
}


MyString MyString::operator+(const char * str)
{
int newsize = this->m_Size + strlen(str) + 1;
char *temp = new char[newsize];
memset(temp, 0, newsize);
strcat(temp, this->pString);
strcat(temp, str);

MyString newstring(temp);
delete[] temp;

return newstring;
}

MyString MyString::operator+(const MyString& str)
{
int newsize = this->m_Size + str.m_Size + 1;
char *temp = new char[newsize];
memset(temp, 0, newsize);
strcat(temp, this->pString);
strcat(temp, str.pString);

MyString newstring(temp);
```

```
delete[] temp;
return newstring;
}

bool MyString::operator==(const char * str)
{
if (strcmp(this->pString, str) == 0 && strlen(str) == this->m_Size){
return true;
}

return false;
}

bool MyString::operator==(const MyString& str)
{
if (strcmp(this->pString, str.pString) == 0 && str.m_Size == this->m_Siz
e){
return true;
}

return false;
}
TestMyString.cpp
void test01()
{
MyString str("hello World");

cout << str << endl;

//cout << "请输入MyString类型字符串: " << endl;
//cin >> str;

//cout << "字符串为:  " << str << endl;

//测试[]
cout << "MyString的第一个字符为: " << str[0] << endl;

//测试 =
MyString str2 = "^_^";
MyString str3 = "";
str3 = "aaaa";
```

```
180 str3 = str2;
181 cout << "str2 = " << str2 << endl;
182 cout << "str3 = " << str3 << endl;
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
1	[]	数组下标	数组名[常量表达式]	左到右	--
	()	圆括号	(表达式）/函数名(形参表)		--
	.	成员选择（对象）	对象.成员名		--
	->	成员选择（指针）	对象指针->成员名		--
2	-	负号运算符	-表达式	右到左	单目运算符
	~	按位取反运算符	~表达式		
	++	自增运算符	++变量名/变量名++		
	--	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		
	!	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)		--
3	/	除	表达式/表达式	左到右	双目运算符
	*	乘	表达式*表达式		
	%	余数（取模）	整型表达式%整型表达式		
4	+	加	表达式+表达式	左到右	双目运算符
	-	减	表达式-表达式		
5	<<	左移	变量<<表达式	左到右	双目运算符
	>>	右移	变量>>表达式		

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	左到右	双目运算符
	! =	不等于	表达式!= 表达式		

8	&	按位与	表达式&表达式	左到右	双目运算符
9	^	按位异或	表达式^表达式	左到右	双目运算符
10	|	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
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

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

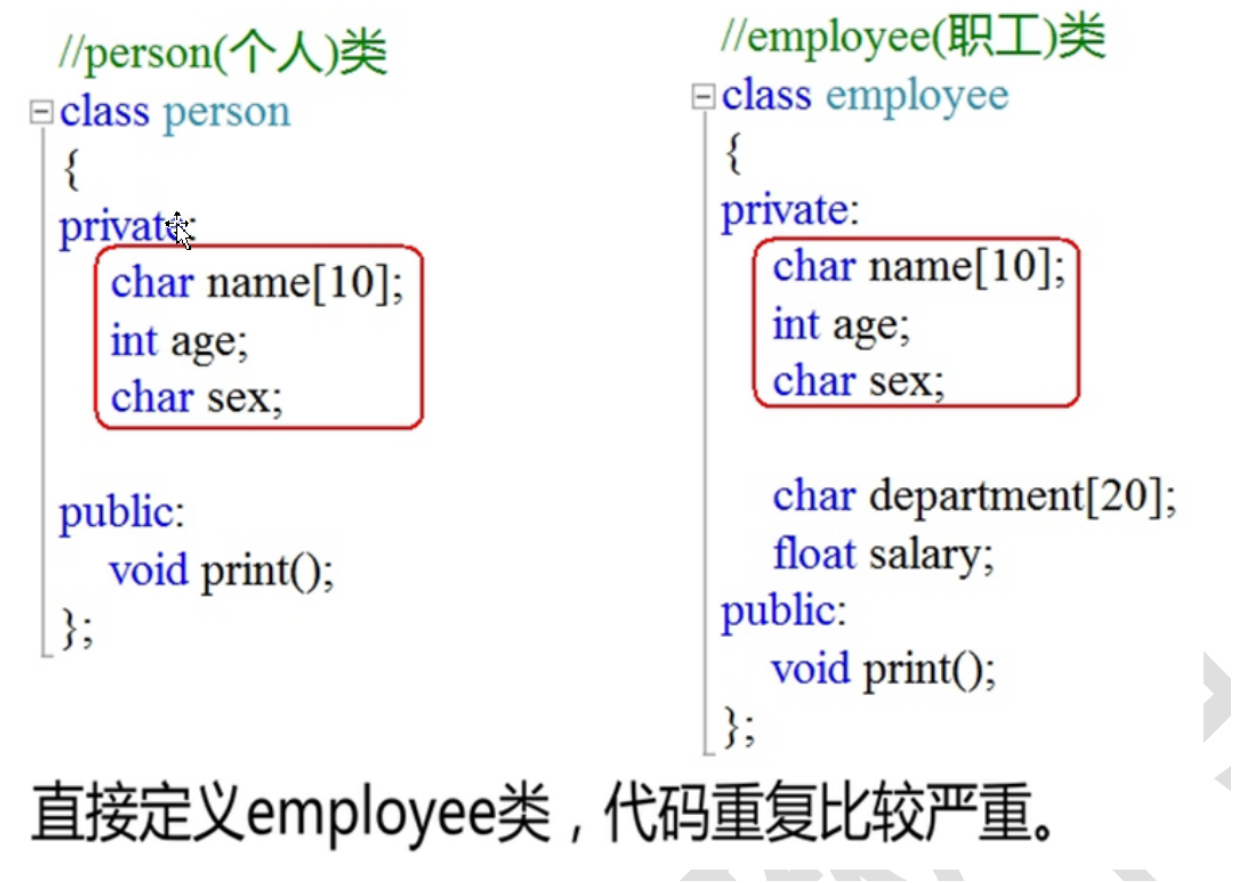
	-=	减后赋值	变量-=表达式		--
	<<=	左移后赋值	变量<<=表达式		--
	>>=	右移后赋值	变量>>=表达式		--
	&=	按位与后赋值	变量&=表达式		--
	^=	按位异或后赋值	变量^=表达式		--
	|=	按位或后赋值	变量|=表达式		--
15	,	逗号运算符	表达式,表达式,...	左到右	--

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



1.2 继承的概念

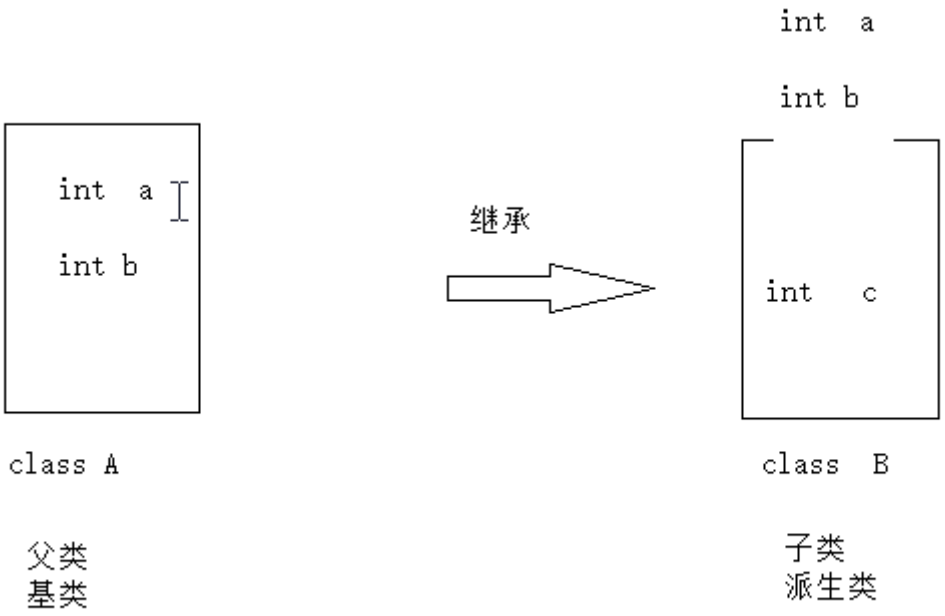
C++最重要的特征是代码重用，通过继承机制可以利用已有的数据类型来定义新的数据类型，新的类不仅拥有旧类的成员，还拥有新定义的成员。

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

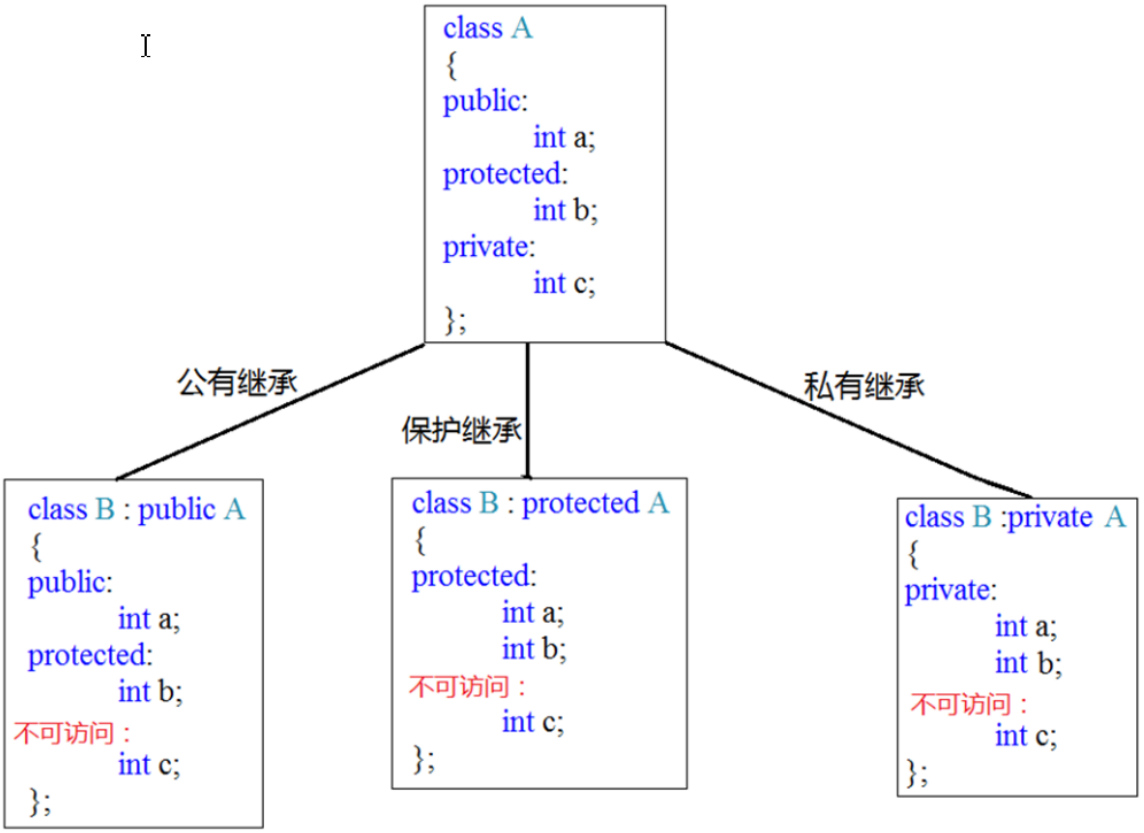
private ： 私有继承

protected ： 保护继承

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
// 基类
class Animal
{
 public:
  int age;
  void print()
  {
  cout << age << endl;
  }
```

```
};
class Dog :public Animal
{
public:
 int tail_len;

 /*
 int age;
 void print()
 {
 cout << age << endl;
 }
 */
};

void test01()
{
 Dog d;


}

int main()
{

 return 0;
}
```

2 派生类访问权限控制




```cpp
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class Base
{
public:
  int a;
 protected:
  int b;
 private:
  int c;
};
//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
class A :public Base
{
 /*
 public:
 int a;
```

```
 protected:
 int b;
 private:
 int c;
 */
public:
 int d;
 void show()
 {
 //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
 //cout << a << b << c << endl;
 }

};
class B :protected Base
{
public:
 /*
 //公有继承 将父类中的公有的权限变成保护的,其他不变
 protected:
 int a;
 protected:
 int b;
 private:
 int c;
 */


 int d;
 void show()
 {
 //子类访问父类 不能访问父类的私有成员
 //cout << a << b << c << d << endl;
 }
};

class C : private Base
{
 //私有继承 会将所有的权限都变成私有的
 /*
```

```
 private:
 int a;
 private:
 int b;
 private:
 int c;
 */
public:
 int d;
 void show()
 {
 //cout << a << b << c << d << endl;
 }

};
void test01()
{
 A p;
 //p通过类外可以访问公有的权限
 p.a = 10;
 p.d = 20;
 B p1;
 //p1.a = 100;
 p1.d = 200;

 C p2;
 //p2.a = 100;
 p2.d = 100;

}


int main()
{

 return 0;
}
```

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：

```
class Aclass{
public:
int mA;
int mB;
};
class Bclass : public Aclass{
public:
int mC;
};
 class Cclass : public Bclass{
 public:
 int mD;
 };
 void test(){
 cout << "A size:" << sizeof(Aclass) << endl;//8
 cout << "B size:" << sizeof(Bclass) << endl;//12
 cout << "C size:" << sizeof(Cclass) << endl;//16
 }
```

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)
```

```
 {
 this->age = age;
 this->name = name;
 cout << "Base的构造函数" << endl;
 }
 ~Base()
 {
 cout << "Base的析构函数" << endl;
 }
 int age;
 string name;

};
//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
class Son:public Base
{
public:
 Son(int id,int age,string name):Base(age, name)
 {
 this->id = id;
 cout << "Son 的构造函数" << endl;
 }
 ~Son()
 {
 cout << "Son 的析构函数" << endl;
 }
 int id;

};
void test01()
{
 Son p(10,18,"lucy");

}
int main()
{
 test01();
 return 0;
}

```

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

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class Base
{
public:
 Base(int a)
 {
 this->a = a;
 }
 void foo()
 {
 cout << "父类的foo函数" << endl;
 }
 int a;
};
class Son :public Base
{
public:
 Son(int a1, int a2):Base(a1),a(a2)
 {}
 void foo()
 {
 cout << "子类的foo函数" << endl;
 }
 int a;

};
void test01()
{
 Son p(10, 20);
 //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
```

```
 //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
 cout << p.a << endl;
 p.foo();

}
int main()
{
 test01();
 return 0;
}
```

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

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

class Base {
public:
  static int getNum() { return sNum; }
  static int getNum(int param) {
  return sNum + param;
  }
public:
  static int sNum;
};
int Base::sNum = 10;

class Derived : public Base {
public:
  static int sNum; //基类静态成员属性将被隐藏
#if 0
```

```
 //重定义一个函数，基类中重载的函数被隐藏
 static int getNum(int param1, int param2) {
 return sNum + param1 + param2;
 }
#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
 static void getNum(int param1, int param2) {
 cout << sNum + param1 + param2 << endl;
 }
#endif
};
int Derived::sNum = 20;
void test01()
{
 Derived p1;
 //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;
 //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
 p1.getNum(1,2);

}

int main()
{
 test01();
 return 0;
}


```

7 多继承

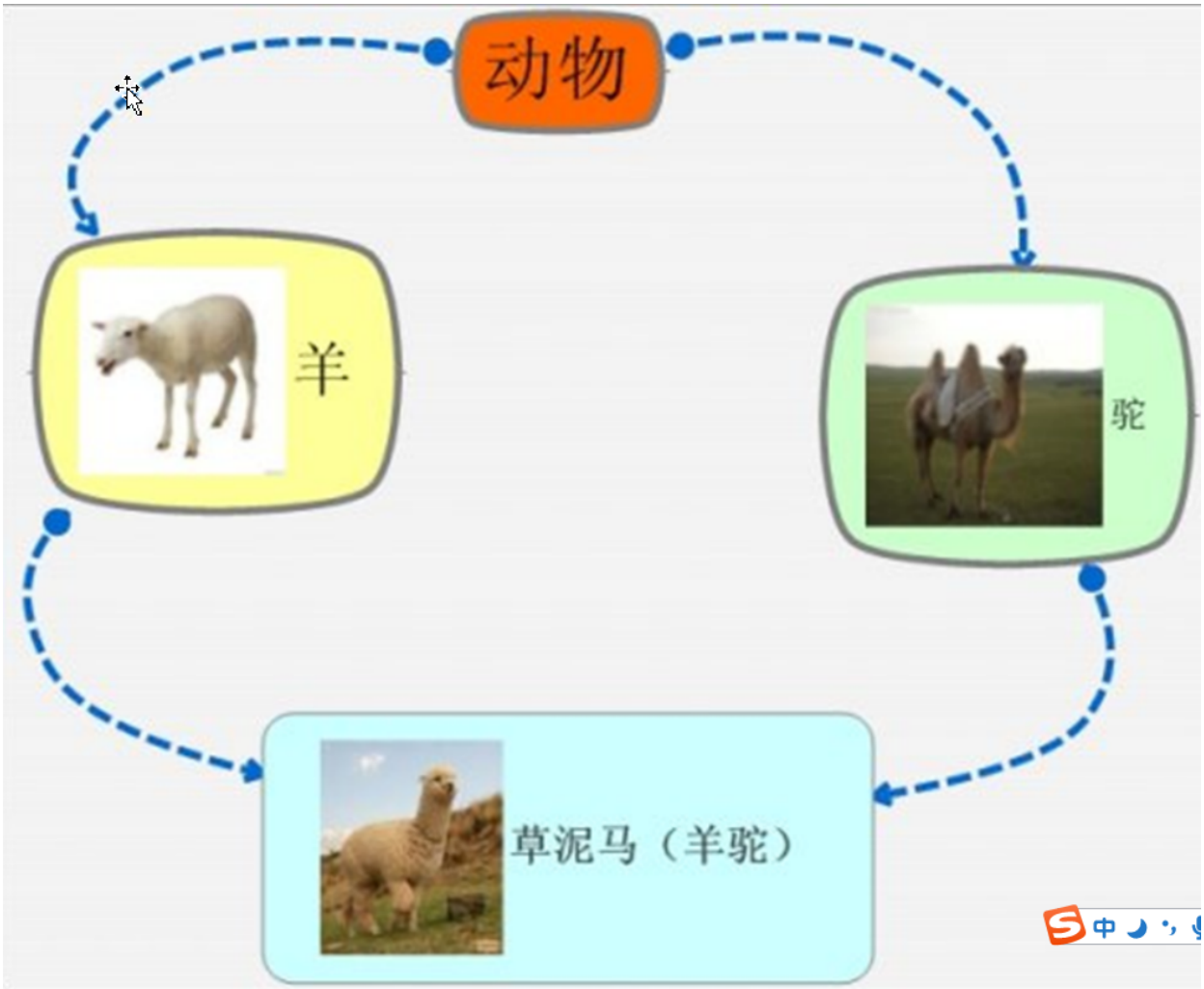
7.1 多继承的概念

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;
```

```
7
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()
29 {
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




```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class animal
{
public:
  int age;
};
class sheep:virtual public animal
{
public:
 int id;
};
class camel:virtual public animal
{
public:
```

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

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class Animal
{
public:
 virtual void speak() //虚函数
 {
 cout << "动物在说话" << endl;
 }

};

class Dog:public Animal
{
public:
 //重写虚函数 函数的返回值 参数 函数名一致
 void speak()
 {
 cout << "狗在说话" << endl;
 }

};

class Cat :public Animal
{
public:
 void speak()
 {
 cout << "猫在说话" << endl;
```

```
 }

};
//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)
{
 obj.speak();//地址早绑定 -> 加上函数前面加上virtual关键字 地址晚绑定
}

void test01()
{
 Animal p1;
 do_work(p1);

 Dog p2;
 do_work(p2);

 Cat p3;
 do_work(p3);
}
int main()
{
 test01();
 return 0;
}
```

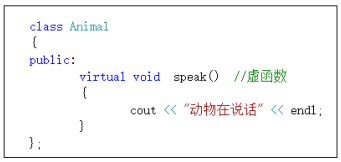
2 多态实现计算器的案例

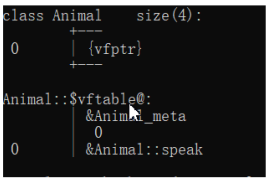
```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;

//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 == "+")
 {
 return a + b;
 }
 else if (cmd == "-")
 {
 return a - b;
 }
 else if (cmd == "*")
 {
 return a * b;
 }
 }
};
void test01()
{
 Mycalc p;
 cout << p.calc(3, 4, "+") << endl;
 cout << p.calc(3, 4, "-") << endl;
}
/*****************************************************************/
//多态实现计算器案例
class Calc
{
public:
 virtual int mycalc(int a, int b)
 {
 return 0;
 }
};

class Add :public Calc
{
public:

 int mycalc(int a, int b)
 {
 return a + b;
 }
};
```

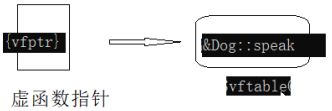
```
54 class Sub :public Calc
55 {
56 public:
57  int mycalc(int a, int b)
58  {
59  return a - b;
60  }
61 };
62 class Mul :public Calc
63 {
64 public:
65  int mycalc(int a, int b)
66  {
67  return a * b;
68  }
69 };
70
71 int do_calc(int a, int b, Calc &obj)
72 {
73
74  return obj.mycalc(a, b);
75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

3 c++如何实现动态绑定

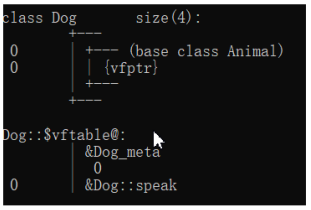




Dog p;//当定义Dog类型对象
时会先构造父类的对象



虚函数指针



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

```
1
2  //多态实现计算器案例
3  class Calc
4  {
5  public:
6   virtual int mycalc(int a, int b) = 0;//虚函数等于0 纯虚函数
7   /*{
8   return 0;
9   }*/
10  };
11  class Mod:public Calc
12  {
13  public:
14   //子类继承了抽象类,那么子类也是一个抽象类
15   int mycalc(int a, int b){}//如果子类重写类虚函数 就不是抽象类
16  };
17
18  //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  {
21   //Calc p;
22   Mod p1;
23  }
```

```
24
```

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:
10   virtual void speak() //虚函数
11   {
12   cout << "动物在说话" << endl;
13   }
14   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
数
15   {
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()
```

```
 {
 cout << "狗在说话" << endl;
 }
 ~Dog()
 {
 cout << "狗的析构" << endl;
 }

};

void do_work(Animal &obj)
{
 obj.speak();//地址早绑定 -> 加上函数前面加上virtual关键字 地址晚绑定
}
void test01()
{
 Animal *p = new Dog;
 p->speak();
 delete p;

}
int main()
{
 test01();
 return 0;
}
```

7 纯虚析构

虚析构函数等于0

```
class Animal
{
public:
 virtual void speak() //虚函数
 {
 cout << "动物在说话" << endl;
 }
 virtual ~Animal() = 0;//纯虚析构
 /*{
```

```
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