

程序设计实习

C++ 面向对象程序设计

张勤健
zqj@pku.edu.cn

北京大学信息科学技术学院

2025 年 2 月 19 日

大纲

- 1 类成员的可访问范围
- 2 【前置知识点补充 3】：函数重载、函数缺省参数
- 3 成员函数的重载及参数缺省
- 4 构造函数
- 5 复制构造函数
- 6 转换构造函数
- 7 析构函数
- 8 构造函数和析构函数什么时候被调用？

类成员的可访问范围

结构化程序设计 vs 面向对象程序设计

类成员的可访问范围

结构化程序设计 vs 面向对象程序设计 -> 封装

类成员的可访问范围

结构化程序设计 vs 面向对象程序设计 -> 封装

在类的定义中，用下列访问范围关键字来说明类成员可被访问的范围：

- `private`: 私有成员，只能在成员函数内访问

类成员的可访问范围

结构化程序设计 vs 面向对象程序设计 -> 封装

在类的定义中，用下列访问范围关键字来说明类成员可被访问的范围：

- `private`: 私有成员，只能在成员函数内访问
- `public`: 公有成员，可以在任何地方访问

类成员的可访问范围

结构化程序设计 vs 面向对象程序设计 -> 封装

在类的定义中，用下列访问范围关键字来说明类成员可被访问的范围：

- `private`: 私有成员，只能在成员函数内访问
- `public`: 公有成员，可以在任何地方访问
- `protected`: 保护成员，以后再说

类成员的可访问范围

结构化程序设计 vs 面向对象程序设计 -> 封装

在类的定义中，用下列访问范围关键字来说明类成员可被访问的范围：

- **private**: 私有成员，只能在成员函数内访问
- **public**: 公有成员，可以在任何地方访问
- **protected**: 保护成员，以后再说

```
1  class className {  
2      private:  
3          //私有属性和函数  
4      public:  
5          //公有属性和函数  
6      protected:  
7          //保护属性和函数  
8  };
```


类成员的可访问范围

结构化程序设计 vs 面向对象程序设计 -> 封装

在类的定义中，用下列访问范围关键字来说明类成员可被访问的范围：

- **private**: 私有成员，只能在成员函数内访问
- **public**: 公有成员，可以在任何地方访问
- **protected**: 保护成员，以后再说

```
1  class className {  
2      private:  
3          //私有属性和函数  
4      public:  
5          //公有属性和函数  
6      protected:  
7          //保护属性和函数  
8  };
```

以上三种关键字出现的次数和先后次序都没有限制。

类成员的可访问范围

如过某个成员前面没有上述关键字，则缺省地被认为是私有成员。

类成员的可访问范围

如过某个成员前面没有上述关键字，则缺省地被认为是私有成员。

```
1  class Man {  
2      int nAge;           // 私有成员  
3      char szName[20];    // 私有成员  
4  public:  
5      void SetName(char *szName) {  
6          strcpy(Man::szName, szName);  
7      }  
8  };
```

类成员的可访问范围

在类的成员函数内部，能够访问：

- 当前对象的全部属性、函数；
- 同类其它对象的全部属性、函数。

类成员的可访问范围

在类的成员函数内部，能够访问：

- 当前对象的全部属性、函数；
- 同类其它对象的全部属性、函数。

在类的成员函数以外的地方，只能够访问该类对象的公有成员。

类成员的可访问范围

```
5  class CEmployee {
6  private:
7      char szName[30]; //名字
8  public:
9      int salary; //工资
10     void setName(const char *name);
11     void getName(char *name);
12     void averageSalary(CEmployee e1, CEmployee e2);
13 };
14 void CEmployee::setName(const char *name) {
15     strcpy(szName, name); // ok
16 }
17 void CEmployee::getName(char *name) {
18     strcpy(name, szName); // ok
19 }
20 void CEmployee::averageSalary(CEmployee e1, CEmployee e2) {
21     cout << e1.szName; // ok, 访问同类其他对象私有成员
22     salary = (e1.salary + e2.salary) / 2;
23 }
```

类成员的可访问范围

```
25  int main() {  
26      CEmployee e;  
27      strcpy(e.szName, "Tom1234567889"); //编译错, 不能访问私有成员  
28      e.setName("Tom");                  // ok  
29      e.salary = 5000;                   // ok  
30      return 0;  
31  }
```

类成员的可访问范围

```
25 int main() {  
26     CEmployee e;  
27     strcpy(e.szName, "Tom1234567889"); //编译错，不能访问私有成员  
28     e.setName("Tom");                  // ok  
29     e.salary = 5000;                   // ok  
30     return 0;  
31 }
```

设置私有成员的机制，叫“隐藏”

“隐藏”的目的是强制对成员变量的访问一定要通过成员函数进行，那么以后成员变量的类型等属性修改后，只需要更改成员函数即可。否则，所有直接访问成员变量的语句都需要修改。

“隐藏”的作用

移植问题：

如果将上面的程序移植到内存空间紧张的手持设备上，希望将szName改为char szName[5]

“隐藏”的作用

移植问题：

如果将上面的程序移植到内存空间紧张的手持设备上，希望将szName改为char szName[5]

- 若szName不是私有，那么就要找出所有类
似strcpy(e.szName, "Tom1234567889");这样的语句进行修改，以防止数组越界。
这样做很麻烦。
- 如果将szName变为私有，那么程序中就不可能出现（除非在类的内部）
strcpy(e.szName, "Tom1234567889");这样的语句，所有对 szName 的访问都是通过成员函数来进行，比如：e.setName("Tom12345678909887");
只要改 setName成员函数，在里面确保不越界就可以了。

用 struct 定义类

```
1 struct CEmployee {  
2     char szName[30]; //公有!!  
3 public:  
4     int salary; //工资  
5     void setName(char *name);  
6     void getName(char *name);  
7     void averageSalary(CEmployee e1, CEmployee e2);  
8 };
```

用 struct 定义类

```
1 struct CEmployee {  
2     char szName[30]; //公有!!  
3 public:  
4     int salary; //工资  
5     void setName(char *name);  
6     void getName(char *name);  
7     void averageSalary(CEmployee e1, CEmployee e2);  
8 };
```

和用“`class`”的唯一区别，就是未说明是公有还是私有的成员，就是公有

【前置知识点补充 3】：函数重载

一个或多个函数，名字相同，然而参数个数或参数类型不相同，这叫做函数的重载。

【前置知识点补充 3】：函数重载

一个或多个函数，名字相同，然而参数个数或参数类型不相同，这叫做函数的重载。以下三个函数是重载关系：

```
1  int _max(double f1, double f2) {}  
2  int _max(int n1, int n2) {}  
3  int _max(int n1, int n2, int n3) {}
```

【前置知识点补充 3】：函数重载

一个或多个函数，名字相同，然而参数个数或参数类型不相同，这叫做函数的重载。以下三个函数是重载关系：

```
1  int _max(double f1, double f2) {}  
2  int _max(int n1, int n2) {}  
3  int _max(int n1, int n2, int n3) {}
```

函数重载使得函数命名变得简单。

编译器根据调用语句的中的实参的个数和类型判断应该调用哪个函数。

【前置知识点补充 3】：函数重载

一个或多个函数，名字相同，然而参数个数或参数类型不相同，这叫做函数的重载。以下三个函数是重载关系：

```
1  int _max(double f1, double f2) {}  
2  int _max(int n1, int n2) {}  
3  int _max(int n1, int n2, int n3) {}
```

函数重载使得函数命名变得简单。

编译器根据调用语句的中的实参的个数和类型判断应该调用哪个函数。

```
1  _max(3.4, 2.5); // 调用 (1)  
2  _max(2, 4); // 调用 (2)  
3  _max(1, 2, 3); // 调用 (3)  
4  _max(3, 2.4); // error, 二义性
```


【前置知识点补充 3】：函数的缺省参数

C++ 中，定义函数的时候可以让最右边的连续若干个参数有缺省值，那么调用函数的时候，若相应位置不写参数，参数就是缺省值。

【前置知识点补充 3】：函数的缺省参数

C++ 中，定义函数的时候可以让最右边的连续若干个参数有缺省值，那么调用函数的时候，若相应位置不写参数，参数就是缺省值。

```
1 void func(int x1, int x2 = 2, int x3 = 3) {}
```

```
1 func(10) ; //等效于 func(10,2,3)
2 func(10, 8) ; //等效于 func(10,8,3)
3 func(10, , 8) ; //不行，只能最右边的连续若干个参数缺省
```

【前置知识点补充 3】：函数的缺省参数

函数参数可缺省的目的在于提高程序的可扩充性。

即如果某个写好的函数要添加新的参数，而原先那些调用该函数的语句，未必需要使用新增的参数，那么为了避免对原先那些函数调用语句的修改，就可以使用缺省参数。

不定项选择题

下面的说法不正确的是：

- Ⓐ 多个重载函数的参数个数必须不同。
- Ⓑ 两个函数，参数表相同，返回值类型不同，它们是重载关系。
- Ⓒ 调用一个第二个和第三个参数都有有缺省值的函数时，可以不写第二个实参而写第三个实参。

不定项选择题

下面的说法不正确的是：

- Ⓐ 多个重载函数的参数个数必须不同。
- Ⓑ 两个函数，参数表相同，返回值类型不同，它们是重载关系。
- Ⓒ 调用一个第二个和第三个参数都有缺省值的函数时，可以不写第二个实参而写第三个实参。

答案：ABC

成员函数的重载及参数缺省

- 成员函数也可以重载
- 成员函数可以带缺省参数。

构造函数-基本概念

成员函数的一种：名字与类名相同，可以有参数，不能有返回值 (`void`也不行)

构造函数-基本概念

成员函数的一种：名字与类名相同，可以有参数，不能有返回值 (`void`也不行)

- 作用：对对象进行初始化，如给成员变量赋初值

构造函数-基本概念

成员函数的一种：名字与类名相同，可以有参数，不能有返回值 (`void`也不行)

- 作用：对对象进行初始化，如给成员变量赋初值
 - 不必专门再写初始化函数，也不用担心忘记调用初始化函数。
 - 避免对象没被初始化就使用而导致程序出错。
- 如果定义类时没写构造函数，则编译器生成一个默认无参数的构造函数。默认构造函数无参数，不做任何操作。

构造函数-基本概念

成员函数的一种：名字与类名相同，可以有参数，不能有返回值 (`void`也不行)

- 作用：对对象进行初始化，如给成员变量赋初值
 - 不必专门再写初始化函数，也不用担心忘记调用初始化函数。
 - 避免对象没被初始化就使用而导致程序出错。
- 如果定义类时没写构造函数，则编译器生成一个默认无参数的构造函数。默认构造函数无参数，不做任何操作。
- 如果定义了构造函数，则编译器不生成默认无参数的构造函数

构造函数-基本概念

成员函数的一种：名字与类名相同，可以有参数，不能有返回值 (`void`也不行)

- 作用：对对象进行初始化，如给成员变量赋初值
 - 不必专门再写初始化函数，也不用担心忘记调用初始化函数。
 - 避免对象没被初始化就使用而导致程序出错。
- 如果定义类时没写构造函数，则编译器生成一个默认无参数的构造函数。默认构造函数无参数，不做任何操作。
- 如果定义了构造函数，则编译器不生成默认无参数的构造函数
- 对象生成时构造函数自动被调用。对象一旦生成，就再也不能在其上执行构造函数

构造函数-基本概念

成员函数的一种：名字与类名相同，可以有参数，不能有返回值 (`void`也不行)

- 作用：对对象进行初始化，如给成员变量赋初值
 - 不必专门再写初始化函数，也不用担心忘记调用初始化函数。
 - 避免对象没被初始化就使用而导致程序出错。
- 如果定义类时没写构造函数，则编译器生成一个默认无参数的构造函数。默认构造函数无参数，不做任何操作。
- 如果定义了构造函数，则编译器不生成默认无参数的构造函数
- 对象生成时构造函数自动被调用。对象一旦生成，就再也不能在其上执行构造函数
- 一个类可以有多个构造函数

构造函数

```
1  class Complex {  
2  private:  
3      double real;  
4      double imag;  
5  public:  
6      void Set(double r, double i);  
7  }; //编译器自动生成默认构造函数  
8  
9  Complex c1; //默认构造函数被调用  
10 Complex *pc = new Complex; //默认构造函数被调用
```

构造函数

```
1  class Complex {
2  private:
3      double real;
4      double imag;
5  public:
6      Complex(double r, double i = 0);
7  };
8  Complex::Complex(double r, double i) {
9      real = r;
10     imag = i;
11 }
12 Complex c1;                // error, 缺少构造函数的参数
13 Complex *pc1 = new Complex; // error, 没有参数
14 Complex c2(2);             // OK
15 Complex c3(2, 4), c4(3, 5);
16 Complex *pc2 = new Complex(3, 4);
```

构造函数

可以有多个构造函数，参数个数或类型不同

```
1  class Complex {
2  private:
3      double real;
4      double imag;
5  public:
6      void Set(double r, double i);
7      Complex(double r, double i);
8      Complex(double r);
9      Complex(Complex c1, Complex c2);
10 };
11 Complex::Complex(double r, double i) {
12     real = r, imag = i;
13 }
14 Complex::Complex(double r) {
15     real = r, imag = 0;
16 }
17 Complex::Complex(Complex c1, Complex c2) {
18     real = c1.real + c2.real, imag = c1.imag + c2.imag;
19 }
20 Complex c1(3), c2(1, 0), c3(c1, c2); // c1 = {3, 0}, c2 = {1, 0}, c3 = {4, 0};
```

构造函数

构造函数最好是`public`的，`private`构造函数不能直接用来初始化对象

```
1  class CSample {  
2  private:  
3      CSample() {}  
4  };  
5  int main() {  
6      CSample Obj; // err. 唯一构造函数是 private  
7      return 0;  
8  }
```


构造函数

构造函数最好是`public`的，`private`构造函数不能直接用来初始化对象

```
1  class CSample {  
2  private:  
3      CSample() {}  
4  };  
5  int main() {  
6      CSample Obj; // err. 唯一构造函数是 private  
7      return 0;  
8  }
```

`private`构造函数也是有实际用处的，比如单例模式

单选题

有类 A 如下定义：

```
1  class A {  
2      int v;  
3  public:  
4      A(int n) {  
5          v = n;  
6      }  
7  };
```

下面哪条语句是编译不会出错的？

- ☐ A A a1(3);
- ☐ B A a2;
- ☐ C A *p = new A();

单选题

有类 A 如下定义：

```
1  class A {  
2      int v;  
3  public:  
4      A(int n) {  
5          v = n;  
6      }  
7  };
```

下面哪条语句是编译不会出错的？

- ☒ A A a1(3);
- ☐ B A a2;
- ☐ C A *p = new A();

答案：A

构造函数在数组中的使用

```
4  class CSample{
5      int x;
6  public:
7      CSample() {
8          cout << "Constructor 1 Called" << endl;
9      }
10     CSample(int n) {
11         x = n;
12         cout << "Constructor 2 Called" << endl;
13     }
14 };
15 int main() {
16     CSample array1[2];
17     cout << "step1" << endl;
18     CSample array2[2] = {4, 5};
19     cout << "step2" << endl;
20     CSample array3[2] = {3};
21     cout << "step3" << endl;
22     CSample *array4 = new CSample[2];
23     delete[] array4;
24     return 0;
25 }
```

构造函数在数组中的使用

```
4 class CSample{
5     int x;
6 public:
7     CSample() {
8         cout << "Constructor 1 Called" << endl;
9     }
10    CSample(int n) {
11        x = n;
12        cout << "Constructor 2 Called" << endl;
13    }
14};
15int main() {
16    CSample array1[2];
17    cout << "step1" << endl;
18    CSample array2[2] = {4, 5};
19    cout << "step2" << endl;
20    CSample array3[2] = {3};
21    cout << "step3" << endl;
22    CSample *array4 = new CSample[2];
23    delete[] array4;
24    return 0;
25}
```

输出

```
Constructor 1 Called
Constructor 1 Called
step1
Constructor 2 Called
Constructor 2 Called
step2
Constructor 2 Called
Constructor 1 Called
step3
Constructor 1 Called
Constructor 1 Called
```

构造函数在数组中的使用

```
1  class Test {  
2  public:  
3      Test(int n) {}           //(1)  
4      Test(int n, int m) {}   //(2)  
5      Test() {}               //(3)  
6  };
```

构造函数在数组中的使用

```
1  class Test {  
2  public:  
3      Test(int n) {}           //(1)  
4      Test(int n, int m) {}   //(2)  
5      Test() {}               //(3)  
6  };
```

```
Test array1[3] = {1, Test(1, 2)};
```

构造函数在数组中的使用

```
1  class Test {  
2  public:  
3      Test(int n) {}           //(1)  
4      Test(int n, int m) {}   //(2)  
5      Test() {}               //(3)  
6  };
```

```
Test array1[3] = {1, Test(1, 2)};
```

三个元素分别用 (1),(2),(3) 初始化

构造函数在数组中的使用

```
1  class Test {  
2  public:  
3      Test(int n) {}           //(1)  
4      Test(int n, int m) {}   //(2)  
5      Test() {}               //(3)  
6  };
```

```
Test array1[3] = {1, Test(1, 2)};
```

三个元素分别用 (1),(2),(3) 初始化

```
Test array2[3] = {Test(2, 3), Test(1, 2), 1};
```

构造函数在数组中的使用

```
1  class Test {  
2  public:  
3      Test(int n) {}           //(1)  
4      Test(int n, int m) {}   //(2)  
5      Test() {}               //(3)  
6  };
```

```
Test array1[3] = {1, Test(1, 2)};
```

三个元素分别用 (1),(2),(3) 初始化

```
Test array2[3] = {Test(2, 3), Test(1, 2), 1};
```

三个元素分别用 (2),(2),(1) 初始化

构造函数在数组中的使用

```
1  class Test {  
2  public:  
3      Test(int n) {}           //(1)  
4      Test(int n, int m) {}    //(2)  
5      Test() {}               //(3)  
6  };
```

```
Test array1[3] = {1, Test(1, 2)};
```

三个元素分别用 (1),(2),(3) 初始化

```
Test array2[3] = {Test(2, 3), Test(1, 2), 1};
```

三个元素分别用 (2),(2),(1) 初始化

```
Test *pArray[3] = {new Test(4), new Test(1, 2)};
```

构造函数在数组中的使用

```
1  class Test {  
2  public:  
3      Test(int n) {}           //(1)  
4      Test(int n, int m) {}   //(2)  
5      Test() {}               //(3)  
6  };
```

```
Test array1[3] = {1, Test(1, 2)};
```

三个元素分别用 (1),(2),(3) 初始化

```
Test array2[3] = {Test(2, 3), Test(1, 2), 1};
```

三个元素分别用 (2),(2),(1) 初始化

```
Test *pArray[3] = {new Test(4), new Test(1, 2)};
```

两个元素分别用 (1),(2) 初始化

单选题

假设 A 是一个类的名字，下面的语句生成了几个类 A 的对象？

```
1  A *arr[4] = {new A(), NULL, new A()};
```

- ☐ A 1
- ☐ B 2
- ☐ C 3
- ☐ D 4

单选题

假设 A 是一个类的名字，下面的语句生成了几个类 A 的对象？

```
1  A *arr[4] = {new A(), NULL, new A()};
```

- ☒ A 1
- ☐ B 2
- ☐ C 3
- ☐ D 4

答案：B

复制构造函数-基本概念

只有一个参数, 即对同类对象的引用。

复制构造函数-基本概念

只有一个参数, 即对同类对象的引用。

- 形如 `X::X(X &)` 或 `X::X(const X &)`, 二者选一, 后者能以常量对象作为参数

复制构造函数-基本概念

只有一个参数, 即对同类对象的引用。

- 形如 `X::X(X &)` 或 `X::X(const X &)`, 二者选一, 后者能以常量对象作为参数
- 如果没有定义复制构造函数, 那么编译器生成默认复制构造函数。默认的复制构造函数完成复制功能。
- 如果定义的自己的复制构造函数, 则默认的复制构造函数不存在。
- 不允许有形如 `X::X(X)` 的构造函数。

复制构造函数

```
1  class Complex {  
2  private:  
3      double real;  
4      double imag;  
5  };  
6  Complex c1;      //调用缺省无参构造函数  
7  Complex c2(c1);  //调用缺省的复制构造函数, 将 c2 初始化成和 c1 一样
```

复制构造函数

```
1  class Complex {
2  private:
3      double real;
4      double imag;
5  };
6  Complex c1;      //调用缺省无参构造函数
7  Complex c2(c1);  //调用缺省的复制构造函数, 将 c2 初始化成和 c1 一样
```

```
1  class Complex {
2  public:
3      double real;
4      double imag;
5      Complex() {}
6      Complex(const Complex &c) {
7          real = c.real;
8          imag = c.imag;
9          cout << "Copy Constructor called";
10     }
11 };
12 Complex c1;
13 Complex c2(c1); //调用自己定义的复制构造函数, 输出 Copy Constructor called
```

复制构造函数起作用的三种情况

① 当用一个对象去初始化同类的另一个对象时。

```
1 Complex c2(c1);  
2 Complex c2 = c1; //初始化语句，非赋值语句
```

复制构造函数起作用的三种情况

- ① 当用一个对象去初始化同类的另一个对象时。

```
1 Complex c2(c1);  
2 Complex c2 = c1; //初始化语句，非赋值语句
```

- ② 如果函数有一个参数是类的对象，那么该函数被调用时，类的复制构造函数将被调用。

```
1 void func(Complex c) {}  
2 int main() {  
3     Complex c1;  
4     func(c1);  
5     return 0;  
6 }
```

复制构造函数起作用的三种情况

- ① 当用一个对象去初始化同类的另一个对象时。

```
1 Complex c2(c1);  
2 Complex c2 = c1; //初始化语句，非赋值语句
```

- ② 如果函数有一个参数是类的对象，那么该函数被调用时，类的复制构造函数将被调用。

```
1 void func(Complex c) {}  
2 int main() {  
3     Complex c1;  
4     func(c1);  
5     return 0;  
6 }
```

- ③ 如果函数的返回值是类的对象时，则函数返回时，类的复制构造函数被调用。

```
1 A func() {  
2     A b(4);  
3     return b;  
4 }  
5 int main() {  
6     cout << func().v << endl;  
7     return 0;  
8 }
```

复制构造函数例题

注意：对象间赋值并不导致复制构造函数被调用

```
4  class CMyclass {
5  public:
6      int n;
7      CMyclass(){};
8      CMyclass(const CMyclass &c) { n = 2 * c.n; }
9  };
10 int main() {
11     CMyclass c1, c2;
12     c1.n = 5;
13     c2 = c1;
14     CMyclass c3(c1);
15     cout << "c2.n=" << c2.n << ", ";
16     cout << "c3.n=" << c3.n << endl;
17     return 0;
18 }
```

复制构造函数例题

注意：对象间赋值并不导致复制构造函数被调用

```
4  class CMyclass {
5  public:
6      int n;
7      CMyclass(){};
8      CMyclass(const CMyclass &c) { n = 2 * c.n; }
9  };
10 int main() {
11     CMyclass c1, c2;
12     c1.n = 5;
13     c2 = c1;
14     CMyclass c3(c1);
15     cout << "c2.n=" << c2.n << ", ";
16     cout << "c3.n=" << c3.n << endl;
17     return 0;
18 }
```

输出:

```
c2.n=5,c3.n=10
```


常量引用参数的使用

```
1 void fun(CMyclass obj) {  
2     cout << "fun" << endl;  
3 }
```

常量引用参数的使用

```
1 void fun(CMyclass obj) {  
2     cout << "fun" << endl;  
3 }
```

这样的函数，调用时生成形参会引发复制构造函数调用，开销比较大。

所以可以考虑使用 CMyclass & 引用类型作为参数。

如果希望确保实参的值在函数中不应被改变，那么可以加上 `const` 关键字

```
1 void fun(const CMyclass &obj) {  
2     //函数中任何试图改变 obj 值的语句都将是变成非法  
3 }
```

假设 A 是一个类的名字，下面哪段程序不会调用 A 的复制构造函数？

- ☐ A `A a1,a2; a1 = a2;`
- ☐ B `void func(A a) { cout << "good" << endl; }`
- ☐ C `A func() { A tmp; return tmp; }`
- ☐ D `A a1; A a2(a1);`

假设 A 是一个类的名字，下面哪段程序不会调用 A 的复制构造函数？

- Ⓐ `A a1,a2; a1 = a2;`
- Ⓑ `void func(A a) { cout << "good" << endl; }`
- Ⓒ `A func() { A tmp; return tmp; }`
- Ⓓ `A a1; A a2(a1);`

答案：A

为什么要自己写复制构造函数？

或者说什么情况下需要自己复制构造函数？
这个稍后再讲

转换构造函数

- 定义转换构造函数的目的是实现类型的自动转换。
- 不以说明符 `explicit` 声明 {且可以用单个参数调用 (C++11 前)} 的构造函数被称为转换构造函数
- 当需要的时候, 编译系统会自动调用转换构造函数, 建立一个无名的临时对象 (或临时变量)。

转换构造函数

```
4  class Complex {
5  public:
6      double real, imag;
7      Complex(int i) { //类型转换构造函数
8          cout << "IntConstructor called" << endl;
9          real = i;
10         imag = 0;
11     }
12     Complex(double r, double i) {
13         real = r;
14         imag = i;
15     }
16 };
17 int main() {
18     Complex c1(7, 8);
19     Complex c2 = 12;
20     c1 = 9; // 9 被自动转换成一个临时 Complex 对象
21     cout << c1.real << "," << c1.imag << endl;
22     return 0;
23 }
```

转换构造函数

```
4  class Complex {
5  public:
6      double real, imag;
7      explicit Complex(int i) { //非类型转换构造函数
8          cout << "IntConstructor called" << endl;
9          real = i, imag = 0;
10     }
11     Complex(double r, double i) {
12         real = r, imag = i;
13     }
14 };
15 int main() {
16     Complex c1(7, 8);
17     Complex c2 = 12;    // error
18     c1 = 9;             // error, 9 不能被自动转换成一个临时 Complex 对象
19     c1 = Complex(9);    // ok
20     cout << c1.real << "," << c1.imag << endl;
21     return 0;
22 }
```


转换构造函数

```
4  class Complex {
5  public:
6      double real, imag;
7      explicit Complex(int i) { //非类型转换构造函数
8          cout << "IntConstructor called" << endl;
9          real = i, imag = 0;
10     }
11     Complex(double r, double i) {
12         real = r, imag = i;
13     }
14 };
15 int main() {
16     Complex c1(7, 8);
17     Complex c2 = 12;    // error
18     c1 = 9;             // error, 9 不能被自动转换成一个临时 Complex 对象
19     c1 = Complex(9);    // ok
20     cout << c1.real << "," << c1.imag << endl;
21     return 0;
22 }
```

explicit 关键字的作用：指定构造函数或 {转换函数 (C++11 起)} 或 {推导指引 (C++17 起)} 为显式，即它不能用于隐式转换和复制初始化。

单选题

```
1  class A {  
2      int v;  
3  public:  
4      A(int i) {  
5          v = i;  
6      }  
7      A() {}  
8  };
```

下面哪段程序不会引发转换构造函数被调用？

- ☒ A A a1(4);
- ☐ B A a3; a3 = 9;
- ☐ C A a2 = 4;
- ☐ D A a1, a2; a1 = a2;

单选题

```
1  class A {  
2      int v;  
3  public:  
4      A(int i) {  
5          v = i;  
6      }  
7      A() {}  
8  };
```

下面哪段程序不会引发转换构造函数被调用？

- ☒ A A a1(4);
- ☐ B A a3; a3 = 9;
- ☐ C A a2 = 4;
- ☐ D A a1, a2; a1 = a2;

答案：D

析构函数

名字与类名相同，在前面加 ‘~’，没有参数和返回值。

- 析构函数对象消亡时即自动被调用。可以定义析构函数来在对象消亡前做善后工作，比如释放分配的空间等。

析构函数

名字与类名相同，在前面加 ‘~’，没有参数和返回值。

- 析构函数对象消亡时即自动被调用。可以定义析构函数来在对象消亡前做善后工作，比如释放分配的空间等。
- 如果定义类时没写析构函数，则编译器生成缺省析构函数。缺省析构函数什么也不做。

析构函数

名字与类名相同，在前面加 ‘~’，没有参数和返回值。

- 析构函数对象消亡时即自动被调用。可以定义析构函数来在对象消亡前做善后工作，比如释放分配的空间等。
- 如果定义类时没写析构函数，则编译器生成缺省析构函数。缺省析构函数什么也不做。
- 如果定义了析构函数，则编译器不生成缺省析构函数。

析构函数

名字与类名相同，在前面加 ‘~’，没有参数和返回值。

- 析构函数对象消亡时即自动被调用。可以定义析构函数来在对象消亡前做善后工作，比如释放分配的空间等。
- 如果定义类时没写析构函数，则编译器生成缺省析构函数。缺省析构函数什么也不做。
- 如果定义了析构函数，则编译器不生成缺省析构函数。
- 一个类最多只能有一个析构函数

```
1  class String {  
2  private:  
3      char *p;  
4  public:  
5      String() {  
6          p = new char[10];  
7      }  
8      ~String();  
9  };  
10 String::~~String() {  
11     delete[] p;  
12 }
```

析构函数和数组

对象数组生命期结束时，对象数组的每个元素的析构函数都会被调用。

```
4  class Ctest {  
5  public:  
6      ~Ctest() {  
7          cout << "destructor called" << endl;  
8      }  
9  };  
10 int main() {  
11     Ctest array[2];  
12     cout << "End Main" << endl;  
13     return 0;  
14 }
```


析构函数和数组

对象数组生命期结束时，对象数组的每个元素的析构函数都会被调用。

```
4  class Ctest {  
5  public:  
6      ~Ctest() {  
7          cout << "destructor called" << endl;  
8      }  
9  };  
10 int main() {  
11     Ctest array[2];  
12     cout << "End Main" << endl;  
13     return 0;  
14 }
```

输出:

```
End Main  
destructor called  
destructor called
```

析构函数和运算符 delete

delete 运算导致析构函数调用。

```
1 Ctest *pTest;  
2 pTest = new Ctest; //构造函数调用  
3 delete pTest;      //析构函数调用
```

析构函数和运算符 delete

delete 运算导致析构函数调用。

```
1 Ctest *pTest;  
2 pTest = new Ctest; //构造函数调用  
3 delete pTest;      //析构函数调用
```

```
1 pTest = new Ctest[3]; //构造函数调用 3 次  
2 delete[] pTest;      //析构函数调用 3 次
```

析构函数和运算符 delete

delete 运算导致析构函数调用。

```
1 Ctest *pTest;  
2 pTest = new Ctest; //构造函数调用  
3 delete pTest;      //析构函数调用
```

```
1 pTest = new Ctest[3]; //构造函数调用 3 次  
2 delete[] pTest;      //析构函数调用 3 次
```

若new一个对象数组，那么用delete释放时应该写 []。否则只delete一个对象 (调用一次析构函数)

析构函数在对象作为函数返回值返回后被调用

```
4  class CMyclass {
5  public:
6      ~CMyclass() {
7          cout << "destructor" << endl;
8      }
9  };
10 CMyclass obj;
11 CMyclass fun(CMyclass sobj) { //参数对象消亡也会导致析构函数被调用
12     return sobj; //函数调用返回时生成临时对象返回
13 }
14 int main() {
15     obj = fun(obj); //函数调用的返回值（临时对象）被用过后，该临时对象析构函数被调用
16     return 0;
17 }
```

析构函数在对象作为函数返回值返回后被调用

```
4  class CMyclass {
5  public:
6      ~CMyclass() {
7          cout << "destructor" << endl;
8      }
9  };
10 CMyclass obj;
11 CMyclass fun(CMyclass sobj) { //参数对象消亡也会导致析构函数被调用
12     return sobj; //函数调用返回时生成临时对象返回
13 }
14 int main() {
15     obj = fun(obj); //函数调用的返回值（临时对象）被用过后，该临时对象析构函数被调用
16     return 0;
17 }
```

输出:

```
destructor
destructor
destructor
```

为什么要自己写复制构造函数？

- 静态数据成员
- 浅拷贝 vs 深拷贝

构造函数和析构函数什么时候被调用？

输出结果:

```
4  class Demo {
5      int id;
6  public:
7      Demo(int i) {
8          id = i;
9          cout << "id=" << id << " constructed" << endl;
10     }
11     ~Demo() {
12         cout << "id=" << id << " destructed" << endl;
13     }
14 };
15 Demo d1(1);
16 void Func() {
17     static Demo d2(2);
18     Demo d3(3);
19     cout << "func" << endl;
20 }
21 int main() {
22     Demo d4(4);
23     d4 = 6;
24     cout << "main" << endl;
25     {
26         Demo d5(5);
27     }
28     Func();
29     cout << "main ends" << endl;
30     return 0;
31 }
```


构造函数和析构函数什么时候被调用？

输出结果:

id=1 constructed

```
4  class Demo {
5      int id;
6  public:
7      Demo(int i) {
8          id = i;
9          cout << "id=" << id << " constructed" << endl;
10     }
11     ~Demo() {
12         cout << "id=" << id << " destructed" << endl;
13     }
14 };
15 Demo d1(1);
16 void Func() {
17     static Demo d2(2);
18     Demo d3(3);
19     cout << "func" << endl;
20 }
21 int main() {
22     Demo d4(4);
23     d4 = 6;
24     cout << "main" << endl;
25     {
26         Demo d5(5);
27     }
28     Func();
29     cout << "main ends" << endl;
30     return 0;
31 }
```

构造函数和析构函数什么时候被调用？

输出结果:

id=1 constructed

id=4 constructed

```
4 class Demo {
5     int id;
6 public:
7     Demo(int i) {
8         id = i;
9         cout << "id=" << id << " constructed" << endl;
10    }
11    ~Demo() {
12        cout << "id=" << id << " destructed" << endl;
13    }
14 };
15 Demo d1(1);
16 void Func() {
17     static Demo d2(2);
18     Demo d3(3);
19     cout << "func" << endl;
20 }
21 int main() {
22     Demo d4(4);
23     d4 = 6;
24     cout << "main" << endl;
25     {
26         Demo d5(5);
27     }
28     Func();
29     cout << "main ends" << endl;
30     return 0;
31 }
```

构造函数和析构函数什么时候被调用？

输出结果:

id=1 constructed

id=4 constructed

id=6 constructed

```
4  class Demo {
5      int id;
6  public:
7      Demo(int i) {
8          id = i;
9          cout << "id=" << id << " constructed" << endl;
10     }
11     ~Demo() {
12         cout << "id=" << id << " destructed" << endl;
13     }
14 };
15 Demo d1(1);
16 void Func() {
17     static Demo d2(2);
18     Demo d3(3);
19     cout << "func" << endl;
20 }
21 int main() {
22     Demo d4(4);
23     d4 = 6;
24     cout << "main" << endl;
25     {
26         Demo d5(5);
27     }
28     Func();
29     cout << "main ends" << endl;
30     return 0;
31 }
```

构造函数和析构函数什么时候被调用？

输出结果:

id=1 constructed

id=4 constructed

id=6 constructed

id=6 destructed

```
4  class Demo {
5      int id;
6  public:
7      Demo(int i) {
8          id = i;
9          cout << "id=" << id << " constructed" << endl;
10     }
11     ~Demo() {
12         cout << "id=" << id << " destructed" << endl;
13     }
14 };
15 Demo d1(1);
16 void Func() {
17     static Demo d2(2);
18     Demo d3(3);
19     cout << "func" << endl;
20 }
21 int main() {
22     Demo d4(4);
23     d4 = 6;
24     cout << "main" << endl;
25     {
26         Demo d5(5);
27     }
28     Func();
29     cout << "main ends" << endl;
30     return 0;
31 }
```

构造函数和析构函数什么时候被调用？

```
4  class Demo {
5      int id;
6  public:
7      Demo(int i) {
8          id = i;
9          cout << "id=" << id << " constructed" << endl;
10     }
11     ~Demo() {
12         cout << "id=" << id << " destructed" << endl;
13     }
14 };
15 Demo d1(1);
16 void Func() {
17     static Demo d2(2);
18     Demo d3(3);
19     cout << "func" << endl;
20 }
21 int main() {
22     Demo d4(4);
23     d4 = 6;
24     cout << "main" << endl;
25     {
26         Demo d5(5);
27     }
28     Func();
29     cout << "main ends" << endl;
30     return 0;
31 }
```

输出结果:

id=1 constructed

id=4 constructed

id=6 constructed

id=6 destructed

main

构造函数和析构函数什么时候被调用？

```
4  class Demo {
5      int id;
6  public:
7      Demo(int i) {
8          id = i;
9          cout << "id=" << id << " constructed" << endl;
10     }
11     ~Demo() {
12         cout << "id=" << id << " destructed" << endl;
13     }
14 };
15 Demo d1(1);
16 void Func() {
17     static Demo d2(2);
18     Demo d3(3);
19     cout << "func" << endl;
20 }
21 int main() {
22     Demo d4(4);
23     d4 = 6;
24     cout << "main" << endl;
25     {
26         Demo d5(5);
27     }
28     Func();
29     cout << "main ends" << endl;
30     return 0;
31 }
```

输出结果:

id=1 constructed

id=4 constructed

id=6 constructed

id=6 destructed

main

id=5 constructed

构造函数和析构函数什么时候被调用？

```
4  class Demo {
5      int id;
6  public:
7      Demo(int i) {
8          id = i;
9          cout << "id=" << id << " constructed" << endl;
10     }
11     ~Demo() {
12         cout << "id=" << id << " destructed" << endl;
13     }
14 };
15 Demo d1(1);
16 void Func() {
17     static Demo d2(2);
18     Demo d3(3);
19     cout << "func" << endl;
20 }
21 int main() {
22     Demo d4(4);
23     d4 = 6;
24     cout << "main" << endl;
25     {
26         Demo d5(5);
27     }
28     Func();
29     cout << "main ends" << endl;
30     return 0;
31 }
```

输出结果:

id=1 constructed

id=4 constructed

id=6 constructed

id=6 destructed

main

id=5 constructed

id=5 destructed

构造函数和析构函数什么时候被调用？

```
4  class Demo {
5      int id;
6  public:
7      Demo(int i) {
8          id = i;
9          cout << "id=" << id << " constructed" << endl;
10     }
11     ~Demo() {
12         cout << "id=" << id << " destructed" << endl;
13     }
14 };
15 Demo d1(1);
16 void Func() {
17     static Demo d2(2);
18     Demo d3(3);
19     cout << "func" << endl;
20 }
21 int main() {
22     Demo d4(4);
23     d4 = 6;
24     cout << "main" << endl;
25     {
26         Demo d5(5);
27     }
28     Func();
29     cout << "main ends" << endl;
30     return 0;
31 }
```

输出结果:

id=1 constructed

id=4 constructed

id=6 constructed

id=6 destructed

main

id=5 constructed

id=5 destructed

id=2 constructed

构造函数和析构函数什么时候被调用？

```
4  class Demo {
5      int id;
6  public:
7      Demo(int i) {
8          id = i;
9          cout << "id=" << id << " constructed" << endl;
10     }
11     ~Demo() {
12         cout << "id=" << id << " destructed" << endl;
13     }
14 };
15 Demo d1(1);
16 void Func() {
17     static Demo d2(2);
18     Demo d3(3);
19     cout << "func" << endl;
20 }
21 int main() {
22     Demo d4(4);
23     d4 = 6;
24     cout << "main" << endl;
25     {
26         Demo d5(5);
27     }
28     Func();
29     cout << "main ends" << endl;
30     return 0;
31 }
```

输出结果:

id=1 constructed

id=4 constructed

id=6 constructed

id=6 destructed

main

id=5 constructed

id=5 destructed

id=2 constructed

id=3 constructed

构造函数和析构函数什么时候被调用？

```
4  class Demo {
5      int id;
6  public:
7      Demo(int i) {
8          id = i;
9          cout << "id=" << id << " constructed" << endl;
10     }
11     ~Demo() {
12         cout << "id=" << id << " destructed" << endl;
13     }
14 };
15 Demo d1(1);
16 void Func() {
17     static Demo d2(2);
18     Demo d3(3);
19     cout << "func" << endl;
20 }
21 int main() {
22     Demo d4(4);
23     d4 = 6;
24     cout << "main" << endl;
25     {
26         Demo d5(5);
27     }
28     Func();
29     cout << "main ends" << endl;
30     return 0;
31 }
```

输出结果:

id=1 constructed

id=4 constructed

id=6 constructed

id=6 destructed

main

id=5 constructed

id=5 destructed

id=2 constructed

id=3 constructed

func

构造函数和析构函数什么时候被调用？

```
4  class Demo {
5      int id;
6  public:
7      Demo(int i) {
8          id = i;
9          cout << "id=" << id << " constructed" << endl;
10     }
11     ~Demo() {
12         cout << "id=" << id << " destructed" << endl;
13     }
14 };
15 Demo d1(1);
16 void Func() {
17     static Demo d2(2);
18     Demo d3(3);
19     cout << "func" << endl;
20 }
21 int main() {
22     Demo d4(4);
23     d4 = 6;
24     cout << "main" << endl;
25     {
26         Demo d5(5);
27     }
28     Func();
29     cout << "main ends" << endl;
30     return 0;
31 }
```

输出结果:

id=1 constructed

id=4 constructed

id=6 constructed

id=6 destructed

main

id=5 constructed

id=5 destructed

id=2 constructed

id=3 constructed

func

id=3 destructed

构造函数和析构函数什么时候被调用？

```
4  class Demo {
5      int id;
6  public:
7      Demo(int i) {
8          id = i;
9          cout << "id=" << id << " constructed" << endl;
10     }
11     ~Demo() {
12         cout << "id=" << id << " destructed" << endl;
13     }
14 };
15 Demo d1(1);
16 void Func() {
17     static Demo d2(2);
18     Demo d3(3);
19     cout << "func" << endl;
20 }
21 int main() {
22     Demo d4(4);
23     d4 = 6;
24     cout << "main" << endl;
25     {
26         Demo d5(5);
27     }
28     Func();
29     cout << "main ends" << endl;
30     return 0;
31 }
```

输出结果:

id=1 constructed

id=4 constructed

id=6 constructed

id=6 destructed

main

id=5 constructed

id=5 destructed

id=2 constructed

id=3 constructed

func

id=3 destructed

main ends

构造函数和析构函数什么时候被调用？

```
4  class Demo {
5      int id;
6  public:
7      Demo(int i) {
8          id = i;
9          cout << "id=" << id << " constructed" << endl;
10     }
11     ~Demo() {
12         cout << "id=" << id << " destructed" << endl;
13     }
14 };
15 Demo d1(1);
16 void Func() {
17     static Demo d2(2);
18     Demo d3(3);
19     cout << "func" << endl;
20 }
21 int main() {
22     Demo d4(4);
23     d4 = 6;
24     cout << "main" << endl;
25     {
26         Demo d5(5);
27     }
28     Func();
29     cout << "main ends" << endl;
30     return 0;
31 }
```

输出结果:

id=1 constructed

id=4 constructed

id=6 constructed

id=6 destructed

main

id=5 constructed

id=5 destructed

id=2 constructed

id=3 constructed

func

id=3 destructed

main ends

id=6 destructed

构造函数和析构函数什么时候被调用？

```
4  class Demo {
5      int id;
6  public:
7      Demo(int i) {
8          id = i;
9          cout << "id=" << id << " constructed" << endl;
10     }
11     ~Demo() {
12         cout << "id=" << id << " destructed" << endl;
13     }
14 };
15 Demo d1(1);
16 void Func() {
17     static Demo d2(2);
18     Demo d3(3);
19     cout << "func" << endl;
20 }
21 int main() {
22     Demo d4(4);
23     d4 = 6;
24     cout << "main" << endl;
25     {
26         Demo d5(5);
27     }
28     Func();
29     cout << "main ends" << endl;
30     return 0;
31 }
```

输出结果:

id=1 constructed

id=4 constructed

id=6 constructed

id=6 destructed

main

id=5 constructed

id=5 destructed

id=2 constructed

id=3 constructed

func

id=3 destructed

main ends

id=6 destructed

id=2 destructed

构造函数和析构函数什么时候被调用？

```
4  class Demo {
5      int id;
6  public:
7      Demo(int i) {
8          id = i;
9          cout << "id=" << id << " constructed" << endl;
10     }
11     ~Demo() {
12         cout << "id=" << id << " destructed" << endl;
13     }
14 };
15 Demo d1(1);
16 void Func() {
17     static Demo d2(2);
18     Demo d3(3);
19     cout << "func" << endl;
20 }
21 int main() {
22     Demo d4(4);
23     d4 = 6;
24     cout << "main" << endl;
25     {
26         Demo d5(5);
27     }
28     Func();
29     cout << "main ends" << endl;
30     return 0;
31 }
```

输出结果:

id=1 constructed

id=4 constructed

id=6 constructed

id=6 destructed

main

id=5 constructed

id=5 destructed

id=2 constructed

id=3 constructed

func

id=3 destructed

main ends

id=6 destructed

id=2 destructed

id=1 destructed

单选题

```
1  int main() {  
2      A *p = new A[2];  
3      A *p2 = new A;  
4      A a;  
5      delete[] p;  
6  }
```

假设A是一个类的名字，下面的程序片段会类A的调用析构函数几次？

- ☒ A 1
- ☐ B 2
- ☐ C 3
- ☐ D 4

单选题

```
1  int main() {  
2      A *p = new A[2];  
3      A *p2 = new A;  
4      A a;  
5      delete[] p;  
6  }
```

假设A是一个类的名字，下面的程序片段会类A的调用析构函数几次？

- ☒ A 1
- ☐ B 2
- ☐ C 3
- ☐ D 4

答案：C

复制构造函数和析构函数的例子

```
1  #include <iostream>
2  using namespace std;
3
4  class CMyclass {
5  public:
6      CMyclass(){};
7      CMyclass(const CMyclass &c) {
8          cout << "copy constructor" << endl;
9      }
10     ~CMyclass() {
11         cout << "destructor" << endl;
12     }
13 };
14 void fun(CMyclass obj_) {
15     cout << "fun" << endl;
16 }
17 CMyclass c;
18 CMyclass Test() {
19     cout << "test" << endl;
20     return c;
21 }
22 int main() {
23     CMyclass c1;
24     fun(c1);
25     Test();
26     return 0;
27 }
```

输出结果:

复制构造函数和析构函数的例子

```
1  #include <iostream>
2  using namespace std;
3
4  class CMyclass {
5  public:
6      CMyclass(){};
7      CMyclass(const CMyclass &c) {
8          cout << "copy constructor" << endl;
9      }
10     ~CMyclass() {
11         cout << "destructor" << endl;
12     }
13 };
14 void fun(CMyclass obj_) {
15     cout << "fun" << endl;
16 }
17 CMyclass c;
18 CMyclass Test() {
19     cout << "test" << endl;
20     return c;
21 }
22 int main() {
23     CMyclass c1;
24     fun(c1);
25     Test();
26     return 0;
27 }
```

输出结果：
copy constructor

复制构造函数和析构函数的例子

```
1  #include <iostream>
2  using namespace std;
3
4  class CMyclass {
5  public:
6      CMyclass(){};
7      CMyclass(const CMyclass &c) {
8          cout << "copy constructor" << endl;
9      }
10     ~CMyclass() {
11         cout << "destructor" << endl;
12     }
13 };
14 void fun(CMyclass obj_) {
15     cout << "fun" << endl;
16 }
17 CMyclass c;
18 CMyclass Test() {
19     cout << "test" << endl;
20     return c;
21 }
22 int main() {
23     CMyclass c1;
24     fun(c1);
25     Test();
26     return 0;
27 }
```

输出结果：
copy constructor
fun

复制构造函数和析构函数的例子

```
1  #include <iostream>
2  using namespace std;
3
4  class CMyclass {
5  public:
6      CMyclass(){};
7      CMyclass(const CMyclass &c) {
8          cout << "copy constructor" << endl;
9      }
10     ~CMyclass() {
11         cout << "destructor" << endl;
12     }
13 };
14 void fun(CMyclass obj_) {
15     cout << "fun" << endl;
16 }
17 CMyclass c;
18 CMyclass Test() {
19     cout << "test" << endl;
20     return c;
21 }
22 int main() {
23     CMyclass c1;
24     fun(c1);
25     Test();
26     return 0;
27 }
```

输出结果：

copy constructor

fun

destructor //参数消亡

复制构造函数和析构函数的例子

```
1  #include <iostream>
2  using namespace std;
3
4  class CMyclass {
5  public:
6      CMyclass(){};
7      CMyclass(const CMyclass &c) {
8          cout << "copy constructor" << endl;
9      }
10     ~CMyclass() {
11         cout << "destructor" << endl;
12     }
13 };
14 void fun(CMyclass obj_) {
15     cout << "fun" << endl;
16 }
17 CMyclass c;
18 CMyclass Test() {
19     cout << "test" << endl;
20     return c;
21 }
22 int main() {
23     CMyclass c1;
24     fun(c1);
25     Test();
26     return 0;
27 }
```

输出结果：

copy constructor

fun

destructor //参数消亡

test

复制构造函数和析构函数的例子

```
1  #include <iostream>
2  using namespace std;
3
4  class CMyclass {
5  public:
6      CMyclass(){};
7      CMyclass(const CMyclass &c) {
8          cout << "copy constructor" << endl;
9      }
10     ~CMyclass() {
11         cout << "destructor" << endl;
12     }
13 };
14 void fun(CMyclass obj_) {
15     cout << "fun" << endl;
16 }
17 CMyclass c;
18 CMyclass Test() {
19     cout << "test" << endl;
20     return c;
21 }
22 int main() {
23     CMyclass c1;
24     fun(c1);
25     Test();
26     return 0;
27 }
```

输出结果:

copy constructor

fun

destructor //参数消亡

test

copy constructor

复制构造函数和析构函数的例子

```
1  #include <iostream>
2  using namespace std;
3
4  class CMyclass {
5  public:
6      CMyclass(){};
7      CMyclass(const CMyclass &c) {
8          cout << "copy constructor" << endl;
9      }
10     ~CMyclass() {
11         cout << "destructor" << endl;
12     }
13 };
14 void fun(CMyclass obj_) {
15     cout << "fun" << endl;
16 }
17 CMyclass c;
18 CMyclass Test() {
19     cout << "test" << endl;
20     return c;
21 }
22 int main() {
23     CMyclass c1;
24     fun(c1);
25     Test();
26     return 0;
27 }
```

输出结果：

copy constructor

fun

destructor //参数消亡

test

copy constructor

destructor // 返回值临时对象消亡

复制构造函数和析构函数的例子

```
1  #include <iostream>
2  using namespace std;
3
4  class CMyclass {
5  public:
6      CMyclass(){};
7      CMyclass(const CMyclass &c) {
8          cout << "copy constructor" << endl;
9      }
10     ~CMyclass() {
11         cout << "destructor" << endl;
12     }
13 };
14 void fun(CMyclass obj_) {
15     cout << "fun" << endl;
16 }
17 CMyclass c;
18 CMyclass Test() {
19     cout << "test" << endl;
20     return c;
21 }
22 int main() {
23     CMyclass c1;
24     fun(c1);
25     Test();
26     return 0;
27 }
```

输出结果：

copy constructor

fun

destructor //参数消亡

test

copy constructor

destructor // 返回值临时对象消亡

destructor // 局部变量消亡

复制构造函数和析构函数的例子

```
1  #include <iostream>
2  using namespace std;
3
4  class CMyclass {
5  public:
6      CMyclass(){};
7      CMyclass(const CMyclass &c) {
8          cout << "copy constructor" << endl;
9      }
10     ~CMyclass() {
11         cout << "destructor" << endl;
12     }
13 };
14 void fun(CMyclass obj_) {
15     cout << "fun" << endl;
16 }
17 CMyclass c;
18 CMyclass Test() {
19     cout << "test" << endl;
20     return c;
21 }
22 int main() {
23     CMyclass c1;
24     fun(c1);
25     Test();
26     return 0;
27 }
```

输出结果:

copy constructor

fun

destructor //参数消亡

test

copy constructor

destructor // 返回值临时对象消亡

destructor // 局部变量消亡

destructor // 全局变量消亡

复制构造函数在不同编译器中的表现

```
3  class A {
4  public:
5      int x;
6      A(int x_) : x(x_) {
7          cout << x << " constructor called" << endl;
8      }
9      A(const A &a) {
10         x = 2 + a.x;
11         cout << "copy called" << endl;
12     }
13     ~A() {
14         cout << x << " destructor called" << endl;
15     }
16 };
17 A f() {
18     A b(10);
19     return b;
20 }
21 int main() {
22     A a(1);
23     a = f();
24 }
```

预期的输出结果:

```
1 constructor called
10 constructor called
copy called
10 destructor called
12 destructor called
12 destructor called
```

复制构造函数在不同编译器中的表现

```
3 class A {
4 public:
5     int x;
6     A(int x_) : x(x_) {
7         cout << x << " constructor called" << endl;
8     }
9     A(const A &a) {
10        x = 2 + a.x;
11        cout << "copy called" << endl;
12    }
13    ~A() {
14        cout << x << " destructor called" << endl;
15    }
16 };
17 A f() {
18     A b(10);
19     return b;
20 }
21 int main() {
22     A a(1);
23     a = f();
24 }
```

实际一些编译器编译后运行的输出结果:

```
1 constructor called
10 constructor called
10 destructor called
10 destructor called
```

复制构造函数在不同编译器中的表现

某些编译器出于优化目的并未生成返回值临时对象。

G++ 编译选项: `-fno-elide-constructors`