

对象的构造、赋值、和释构

在 C++ 中，默认构造函数、拷贝构造函数、赋值操作符、和释构函数是四种特殊的函数。我们已经学习过一些它们的特殊定义方法，和如何被自动地调用。在学习完继承后，在本章里面，我们会学习一些这些函数之间的调用关系，特别是子类与父类的构造函数的调用关系。

1. 构造函数与释构函数调用的时间和顺序关系

先看下面定义类 A。

```
1 class A
2 {
3     int id;
4 public:
5     A(int id) {
6         this->id = id; // (*this).id = id;
7         cout << "A(int) : " << id << endl;
8     }
9
10    A(const A & a) { // use const whenever possible
11        id = a.id;
12        cout << "(A &) : " << id << endl;
13    }
14
15    A & operator = (const A & a) {
16        id = a.id;
17        cout << "A::operator =" << endl;
18    }
19
20    ~A() {
21        cout << "~A() : " << id << endl;
22    }
23 };
```

在类 A 当中，我们定义了两个构造函数，一个赋值操作符，和一个析构函数。这些函数所做的事情都是：输出了它们各自的函数签名。这样，我们就可以通过控制台输出，观察它们当中那些被调用了和它们被调用的先后顺序。另外，我们在对象的构造和析构函数中分别显示了对象的 id，这样我们就可以分辨不同对象各自的构造和析构函数的调用。

在第 6 行中，我们使用了操作符->。他是一个通过对象的地址访问对象的成员的操作符。也就是说 `this->id` 和 `(*this).id` 具有同样的语意。

写拷贝构造函数的时候，如第 10 行，最好尽量把参数的类型定义为常量。这使得我们可以使用常量，如临时变量和函数返回值，来初始化对象。

从下面的 main 函数中我们可以看到构造函数在什么时候被调用。如果一个对象是一个局部变量，那么它所在的内存空间在函数运行之前已经被分配，但是对象的构造函数仅在函数运行到对象所定义的那行的析构才被调用。

<pre>24 int main() { 25 cout << "start" << endl; 26 A a(1); 27 }</pre>	<p>输出</p> <pre>start A(int) : 1 ~A() : 1</pre>
--	--

从程序的输出我们可以看到，对象 a 的构造函数的输出位于输出的第 2 行。也就是说，对象的构造函数是在第 25

行的输出之后才执行的。最后在函数结束前，对象 a 的时候函数被调用了。

下面我们看看函数当中有多个对象的时候，各个对象的构造函数、析构函数调用的先后顺序。

28 int main() {	输出
29 A a1(1);	A(int) : 1
30 A a2(2);	A(int) : 2
31 }	~A() : 2
	~A() : 1

在上例中，我们看到了对象 a1 较先被构造，但是它却较后被析构。实际上，对于同一个函数的局部变量，较先构造的对象总是较后被释放。原因是，后被构造的对象可能依赖于先被构造的对象。例如，a1 是一个文件而 a2 是一个文件处理的操作，a2 的析构函数可能在文件末位添加文件的结束语，那么这个操作必定要在 a1 的析构函数（关闭文件）之前完成。

下面我们看看同一个对象的构造函数与析构函数的对应关系。

32 int main() {	输出
33 int choice;	-1
34 cin >> choice;	A(int) : 1
35 A a1(1);	~A() : 1
36 if (choice < 0) return 0;	
37 A a2(2);	
38 }	

在上例中我们看到，当程序的输入是-1 的时候，第 37 不会被执行，对象 a2 没有被构造。而且，在这中情况下，a2 的析构函数也不会被调用。也就是说，作为局部变量，对象的构造函数有可能不被调用；而且构造函数与析构函数的调用总是成对的：要么都被调用，要么都不被调用。例如，如果一个文件的对象没有被构造，那么对应的文件没有被打开，那么文件不用被关闭，即对象也没有必要被析构。

下例中我们可以看到，作为同一个函数中的局部变量，不同对象共享同一个内存空间时，它们的构造与析构的时间关系。

<pre>39 int main() { 40 A a1(1); 41 { 42 A a2(2); 43 cout << &a2 << endl; 44 } 45 { 46 A a3(3); 47 cout << &a3 << endl; 48 } 49 }</pre>	<p><u>输出</u></p> <p>A(int) : 1 A(int) : 2 0x7fff59ca2b90 ~A() : 2 A(int) : 3 0x7fff59ca2b90 ~A() : 3 ~A() : 1</p>
---	---

在上例中，由于对象 a2 和 a3 处于不重叠的两个程序块 block，因此具有不同作用域 scope 和生命期 lifetime。C++规定：对象在所在的程序块 block 结束的时候就被析构。这使得上例中的对象 a2 和 a3 可以重用同一个内存空间。我们从输出可以看到，第 43 行与第 46 行中输出的 a2 和 a3 的地址

是一样的，并且 a2 的释构先于 a3 的构造。

下例中我们可以看到，一个对象除了可以不被构造，也可能被构造多次。

<pre>50 int main() { 51 A a1(1); 52 for (int i = 0; i < 2; ++ i) { 53 A a2(2); 54 cout << &a2 << endl; 55 } 56 }</pre>	<p>输出</p> <pre>A(int) : 1 A(int) : 2 0x7fff569e2bb0 ~A() : 2 A(int) : 2 0x7fff569e2bb0 ~A() : 2 ~A() : 1</pre>
---	--

在上例中，对象 a2 所在的块被执行了两次。这个时候，对象 a2 分别被构造和释构了两次。实际上，对象 a2 也可以理解为不同的多个对象，它们分别具有不同的生命期 lifetime。那么，a2 对应的多个对象具有同一个内存空间，它们中的前一个对象的释构必须先于后一个对象的构造。

2. 数组中多个对象的构造和释构

首先我们定义一个类 B，这个类与类 A 不同之处在与它提供一个默认构造函数 default constructor。为了使 B 的每个对象都具有不同的 id，我们使用了一个全局变量 nextId，并在构造每个 B 的时候使用 nextId 的值作为 id 的值，然后给 nextId 加一。

```

57 int nextId = 1;
58
59 class B
60 {
61     int id;
62 public:
63     B() {
64         id = nextId;
65         ++ nextId;
66         cout << "B() : " << id << endl;
67     }
68
69     B(const B & b) {
70         id = b.id;
71         cout << "B(B&) : " << id << endl;
72     }
73
74     B & operator = (const B & b) {
75         id = b.id;
76         cout << "B::operator =" << endl;
77     }
78
79     ~B() {
80         cout << "~B() : " << id << endl;
81     }
82 };

```

下例中我们可以看到一个数组中的对象的构造与释构的先后顺序。

<pre> 83 int main() { 84 B array[3]; 85 } </pre>	<p><u>输出</u></p> <pre> B() : 1 B() : 2 B() : 3 ~B() : 3 ~B() : 2 ~B() : 1 </pre>
--	--

在上例中，我们可以看到数组中的元素的默认构造函数会被调用。释构函数会被以相反的顺序调用。如果在 84 行中

定义的是 **A** 的对象的数组，那么编译器将会报错。这是因为类 **A** 没有提供默认拷贝构造函数。

下例中我们可以看到，动态分配的数组中的对象的构造与释构与作为局部变量的数组是不同的。

<pre>86 int main() { 87 B * array1 = new B[2]; 88 delete [] array1; 89 B * array2 = new B[2]; 90 }</pre>	<u>输出</u> B() : 1 B() : 2 ~B() : 2 ~B() : 1 B() : 3 B() : 4
--	---

在上例中，我们可以看到，动态分配的数组 `array1` 的分配和释放都早与 `array2` 的分配。实际上，动态分配的数组中的元素的默认构造函数会在存储空间被分配的时候被调用。另一方面，动态分配的数组中的元素的释构造函数会在存储空间被释放的时候被调用。具体地说，动态分配的数组中的元素的构造和释构分别发生在操作符 `new` 运行后和操作符 `delete` 之前。如果对象没有被释放，对象将不被释构，如 89 行的 `array2` 中的对象。

下面我们讨论一些拷贝构造函数和编译优化的问题。

<pre>91 int main() { 92 A a = A(1); 93 }</pre>	<u>输出</u> A(int) : 1 ~A() : 1
--	-------------------------------------

在上例中，`A(1)` 生成了一个临时对象，这个临时对象的值再被用作构造对象 `a`，即对象 `a` 的拷贝构造函数会被调用，

并且它会以 `A(1)` 作为参数。这里 `A(1)` 由于产生的是一个临时对象，所以如果类 `A` 的构造函数的签名 `A(A &)` 是而不是 `A(const A &)`，那么，在 92 行，将会看到以下的错误：

```
error: no matching function for call to 'A::A(A)'
```

所以，在构造函数中应该尽量使用 `const`。根据上面所说的，我们应该在输出中看到两个对象的构造和释构才对。但是，我们只看到一个对象的构造和释构。这是因为编译优化的介入。编译优化 compilation optimization 是指在编译的时候通过改写程序对程序的运行效率进行优化，其中包括去除多余的代码和变量。在第 92 行中，`A(1)` 产生的临时对象的作用就是用来初始化对象 `a`，所以两个对象可以被优化成只有一个对象的等价形式 `A a(1)`。因此我们看到的结果只有一个对象的构造和释放。

下例中我们将展示如何给数组中的对象定义初始化列表。

```
94 int main() {  
95     A a[2] = { A(1), A(2) };  
96 }
```

```
输出  
A(int) : 1  
A(int) : 2  
~A() : 2  
~A() : 1
```

上例中我们在初始化列表中给出了两个匿名对象，然后数组中的两个对象将调用它们拷贝构造函数来拷贝列表中两个匿名对象的值。与前一个例子相同，编译优化的介入使得程序中只有两个对象。与前一个例子相同的另一个地方是，虽然程序没有调用拷贝构造函数，但是如果类 `A` 的构造函数

的签名 `A(A &)` 是而不是 `A(const A &)`，编译器将会报错。

那么，初始化列表不够长的时候程序会怎样执行呢？

```
97 int main() {
98     A a[3] = { A(1), A(2) };
99 }
```

我们记得，如果数组中的元素是基本数据类型的时候，初始化列表中将会自动补 0。当数组元素是对象的时候，初始化列表中将会自动补充对象，而这些对象会调用它们的默认构造函数来初始化。由于类 `A` 中没有默认构造函数，所以上例会有以下的编译错误提示：

```
error: no matching function for call to 'A::A()'
```

3. 成员对象构造与释构

下例中我们展示如何初始化作为类中的成员的对象。

```
100 class C
101 {
102     A a;
103 public:
104     C(int id) : a(id) {
105         cout << "C(int)" << endl;
106     }
107
108     ~C() {
109         cout << "~C()" << endl;
110     }
111 };
112
113 int main() {
114     C c(1);
115 }
```

输出

```
A(int) : 1
C(int)
~C()
~A() : 1
```

标准 C++ 规定，成员变量是不可以在声明的时候初始化的。如果一个成员对象没有默认构造函数，那么成员对象必须在构造函数的初始化列表中调用它的一个构造函数。上例中第 104 行给出了构造函数的初始化列表的一个例子。其实，如果一个类有成员对象，那么这个类的每个构造函数都会调用各个成员对象的构造函数以构造各个成员。如果一个构造函数没有初始化列表，那么这个构造函数将会调用成员对象的默认构造函数。另外一方面，如果成员对象没有默认构造函数，那么类中的每个构造函数都必须有初始化列表，以告诉编译器如何初始化各个成员对象（以告诉编译器使用什么参数来调用成员对象的那个构造函数）。

由于对象中的成员变量不会依赖于对象本身（假设设计成员变量的类先于设计当前类）。成员变量的构造函数总是早于当前对象的构造函数本身被调用，而成员变量的析构函数总是晚于当前对象的析构函数本身被调用。如上例的输出所示，成员对象的构造函数 **A(int)** 先于 **C(int)** 被调用，而成员对象的析构函数 **~A()** 晚于 **~C()** 被调用。

下例中我们看看多个成员对象的情况。

```
116 class D
117 {
118     A a1;
119     A a2;
120 public:
121     D(int id) : a2(id), a1(id + 1) {
```

```

122         cout << "D(int)" << endl;
123     }
124
125     // It might cause errors to swap a1 and a2
126     D(const D & d) : a1(d.a2), a2(d.a1) {
127         cout << "D(D&)" << endl;
128     }
129
130     ~D() {
131         cout << "~D()" << endl;
132     }
133 };
134
135 int main() {
136     D d(D(1)); // equals D d1(1); D d2(d1) ?
137 }

```

输出

```

A(int) : 2
A(int) : 1
D(int)
~D()
~A() : 1
~A() : 2

```

当类中有多个成员对象的时候，初始化列表中的多个构造函数用逗号隔开。初始化列表的格式为

: 对象名 1(参数列表 1), 对象名 2(参数列表 2)

多个成员对象的构造函数调用的顺序是由成员对象在类中的声明的顺序决定的，而与初始化列表中的顺序无关。例如，在第 118 和 119 行中我们先定义了 a1 然后定义了 a2，所以虽然第 121 行的初始化列表中 a2 先出现，但是 a1 的构造还是先于 a2 的构造。

在 126 行的拷贝构造函数中，我们试图交换 a1 跟 a2 的

值。但由于在 136 行中编译优化的介入，使得 126 行的拷贝构造函数没有被执行。这使得程序的执行结果不同了。错误的原因在于拷贝构造函数：由于编译优化的存在，使得某些拷贝构造函数不被执行，因此如果拷贝构造函数使得拷贝的对象与被拷贝的对象不同，那么优化前后的程序将会有不同的结果。这里我们得到的教训是：拷贝构造函数不要做拷贝以外的事情。

4. 构造函数、析构函数的链式调用

在 C++ 中，每个子类的构造函数都会调用它的父类中的一个构造函数来初始化对象中属于父类的那部分成员变量。子类的构造函数调用父类的构造函数的必要性在于：

- 既然父类中已经定义了构造函数，所以子类中应该重用这些代码。
- 父类中声明为私有的成员变量不能在子类中初始化，必须通过调用父类中给定的初始化方式正确初始化。

```
138 class E : public A
139 {
140 public:
141     E() : A(1) {
142         cout << "E()" << endl;
143     }
144
145     E(int id) : A(id) {
146         cout << "E(int)" << endl;
147     }
148
```

```

149     E(const E & e) : A(e) { // down-casting E->A&
150         cout << "E(E&)" << endl;
151     }
152
153     ~E() { // automatically call ~A()
154         cout << "~E()" << endl;
155     }
156 };
157
158 int main() {
159     E e1;
160 }

```

输出

```

A(int) : 1
E()
~E()
~A() : 1

```

在 141-151 行中的构造函数中分别在构造函数的初始化列表中调用了父类 **A** 中的构造函数。初始化列表的格式为：

：父类名(参数列表), 对象名 1(参数列表 1), 对象名 2(参数列表 2)

注意参数列表中可以同时调用父类的构造函数和成员对象的构造函数。如果一个构造函数中没有指定一个父类的构造函数, 那么这个构造函数将会调用父类的默认构造函数(这时候要求父类中必须具有默认构造函数)。最后, 子类的析构函数会自动地调用父类的析构函数。

在上例中, 不同的构造函数调用了父类中的不同构造函数。其中, 在 149 行中, 把子类的对象的引用 **e** 传给了父类的构造函数。这是一个 down-casting, 因为类 **E** 是类 **A** 的子类。

下例中我们看看既有父类又有成员对象的情况。

<pre>161 class F : public E 162 { 163 private: 164 A a; 165 166 public: 167 F() : a(2) { // calls E() 168 cout << "F()" << endl; 169 } 170 171 F(int id) : a(id), E(id + 1) { 172 cout << "F(int)" << endl; 173 } 174 175 ~F() { 176 cout << "~F()" << endl; 177 } 178 }; 179 180 int main() { 181 F f2(3); 182 }</pre>	<p>输出</p> <pre>A(int) : 4 E(int) A(int) : 3 F(int) ~F() ~A() : 3 ~E() ~A() : 4</pre>
---	--

当类中有多个成员对象的时候，初始化列表中的多个构造函数的调用逗号隔开。父类的构造函数总会在成员变量的构造函数之前调用。所以虽然在 171 行中，成员对象 **a** 的构造函数出现在父类 **E** 的构造函数之前，但是父类 **E** 的构造函数仍然会先调用。

父类 **E** 的也有一个父类（类 **A**）。对于上例，类 **F** 中的属于类 **A** 的成员对象 **a** 的 **id** 是 3 而父类 **E** 的父类的 **id** 是 4。我们可以看到，在父类 **E** 的构造函数调用之前，调用了它的父类 **A** 的构造函数。在 C++ 中，这种每子类的构造函数自动调用父

类的一个构造函数的机制叫做构造函数的链式调用 constructor chaining。同样地，在 C++ 中，每子类的析构函数都会自动调用父类的析构函数，这叫做析构函数的链式调用 destructor chaining。在链式调用的时候，父类的构造函数的调用先于子类的构造函数，而子类的析构函数的调用先于父类的析构函数。

5. C++ 自动合成的函数

C++ 会自动生成（合成）一些函数。下面，我们归纳一下这些函数在什么情况下会被合成，及合成的函数所做的事情。

函数类型	合成的情况	所做的事情
默认构造函数	类中没有定义任何一个构造函数	1.调用父类的默认构造函数 2.调用各个成员对象的默认构造函数
拷贝构造函数	类中没有定义拷贝构造函数	1.调用父类的拷贝构造函数 2.调用各个成员对象的拷贝构造函数
赋值操作符	类中没有定义赋值操作符	1.调用父类的赋值操作符 2.调用各个成员对象的赋值操作符
析构函数	类中没有定义析构函数	1.调用各个成员对象的析构函数 2.调用父类的析构函数

对于大多数的类（没有地址成员变量的类），没有必要定义这些函数。看下面的例子。

```
183 class G : public A
184 {
185 private:
186     B b;
187
188 public:
189     G() : A(1), b() {
190     }
191
```

输出

```
A(int) : 1
B() : 1
(A &) : 1
B(B&) : 1
A::operator =
B::operator =
~B() : 1
~A() : 1
```

```

192      /*
193      G(const G & g) : A(g), b(g.b) {
194      }
195
196      G & operator = (const G & g) {
197          (*this).A::operator =(g);
198          b = g.b;
199      }
200
201      ~G() { // call ~A() and ~B();
202      }
203      */
204 };
205 int main() {
206     G g1;
207     G g2(g1);
208     g1 = g2;
209 }

```

```

~B() : 1
~A() : 1

```

在上例中，自动合成的拷贝构造函数、赋值操作符、和析构函数所做的事情与注释中的程序完全相同。这个例子中必须定义默认构造函数，否则编译器会报错，因为父类 **A** 中没有默认构造函数。

最后，总结一些编程中的好习惯：

- 如果一个类中需要拷贝构造函数（如类中有成员变量，它存储的是动态分配的内存空间），那么这个类也需要赋值操作符和析构函数。
- 如果浅拷贝可以满足，那么不要定义拷贝构造函数、赋值操作符、和析构函数。
- 尽量继承接口：接口中的默认构造函数、拷贝构造函数、赋值操作符、和析构函数都是空的，我们无需考

虑本章中介绍的任何繁琐问题。

习题

1. 用自己的话解释下面的概念

构造函数的链式调用

2. 用自己的话回答下面的问题

把拷贝构造函数的参数的类型定义为常量有什么用？

定义为函数局部变量的对象的构造函数什么时候调用？

定义为函数局部变量的多个对象的构造函数和析构函数调用从时间上说有什么关系？

定义为函数局部变量的对象的构造函数和析构函数调用有什么关系？

定义为函数局部变量的对象的析构函数什么时候调用？

在循环体内定义的对象构造函数和析构函数如何调用？

定义为函数局部变量的对象数组的构造函数和析构函数调用如何？

动态分配的对象和对象数组的构造函数和析构函数调用如何？

成员变量构造与析构的方法？

成员变量构造与析构的时间？

在子类中如何初始化父类中的私有成员？

构造函数，父类的构造函数，成员变量的构造函数的调用
先后顺序？

释构造函数，父类的释构造函数，成员变量的释构造函数的调用
先后顺序？

那些对象函数会在类中被自动合成？

这些对象函数会在什么时候在类中被自动合成？

这些对象函数的自动合成版本会完成什么工作？

那三个成员函数一般同时定义？

这三个函数在一般情况下需要定义吗？

继承自接口的好处？