第十三章 类的构造、析构与赋值函数

构造函数、析构函数与赋值函数是每个类最基本的函数。它们太普通以致让人容易麻痹大意，其实这些貌似简单的函数就象没有顶盖的下水道那样危险。

每个类只有一个析构函数和一个赋值函数，但可以有多个构造函数（包含一个拷贝构造函数，其它的称为普通构造函数）。对于任意一个类 A，如果不想编写上述函数，C++编译器将自动为 A 产生四个缺省的函数，如

```
A(void);                    // 缺省的无参数构造函数
A(const A &a);              // 缺省的拷贝构造函数
~A(void);                   // 缺省的析构函数
A & operate =(const A &a);// 缺省的赋值函数
```

这不禁让人疑惑，既然能自动生成函数，为什么还要程序员编写？

原因如下：

（1）如果使用“缺省的无参数构造函数”和“缺省的析构函数”，等于放弃了自主“初始化”和“清除”的机会，C++发明人 Stroustrup 的好心好意白费了。

（2）“缺省的拷贝构造函数”和“缺省的赋值函数”均采用“位拷贝”而非“值拷贝”的方式来实现，倘若类中含有指针变量，这两个函数注定将出错。

对于那些没有吃够苦头的 C++程序员，如果他说编写构造函数、析构函数与赋值函数很容易，可以不用动脑筋，表明他的认识还比较肤浅，水平有待于提高。

本章以类 String 的设计与实现为例，深入阐述被很多教科书忽视了的道理。String 的结构如下：

```
class String
{
  public:
    String(const char *str = NULL);   // 普通构造函数
    String(const String &other);  // 拷贝构造函数
    ~ String(void);               // 析构函数
    String & operate =(const String &other); // 赋值函数
  private:
    char    *m_data;              // 用于保存字符串
};
```

13.1 构造函数与析构函数的起源

作为比 C 更先进的语言，C++提供了更好的机制来增强程序的安全性。C++编译器

具有严格的类型安全检查功能，它几乎能找出程序中所有的语法问题，这的确帮了程序员的大忙。但是程序通过了编译检查并不表示错误已经不存在了，在“错误”的大家庭里，“语法错误”的地位只能算是小弟弟。级别高的错误通常隐藏得很深，就象狡猾的罪犯，想逮住他可不容易。

根据经验，不少难以察觉的程序错误是由于变量没有被正确初始化或清除造成的，而初始化和清除工作很容易被人遗忘。Stroustrup 在设计 C++语言时充分考虑了这个问题并很好地予以解决：

把对象的初始化工作放在构造函数中，把清除工作放在析构函数中。当对象被创建时，构造函数被自动执行。当对象消亡时，析构函数被自动执行。这下就不用担心忘了对象的初始化和清除工作。

构造函数与析构函数的名字不能随便起，必须让编译器认得出才可以被自动执行。Stroustrup 的命名方法既简单又合理：

让构造函数、析构函数与类同名，由于析构函数的目的与构造函数的相反，就加前缀‘~’以示区别。

除了名字外，构造函数与析构函数的另一个特别之处是没有返回值类型，这与返回值类型为 void 的函数不同。构造函数与析构函数的使命非常明确，就象出生与死亡，光溜溜地来光溜溜地去。如果它们有返回值类型，那么编译器将不知所措。为了防止节外生枝，干脆规定没有返回值类型。（以上典故参考了文献[Eekel, p55-p56]）

12.2 构造函数的初始化表

构造函数有个特殊的初始化方式叫“初始化表达式表”（简称初始化表）。初始化表位于函数参数表之后，却在函数体 {} 之前。这说明该表里的初始化工作发生在函数体内的任何代码被执行之前。

构造函数初始化表的使用规则：

◆ 如果类存在继承关系，派生类必须在其初始化表里调用基类的构造函数。

例如

```
class A
{...
    A(int x);       // A 的构造函数
};

class B : public A
{...
    B(int x, int y);// B 的构造函数
};

B::B(int x, int y)
 : A(x)             // 在初始化表里调用 A 的构造函数
```

```
{
  …
}
```

◆ 类的 const 常量只能在初始化表里被初始化，因为它不能在函数体内用赋值的方式来初始化（参见 12.4 节）。

◆ 类的数据成员的初始化可以采用初始化表或函数体内赋值两种方式，这两种方式的效率不完全相同。

非内部数据类型的成员对象应当采用第一种方式初始化，以获取更高的效率。例如

```
class A
{…
    A(void);                // 无参数构造函数
    A(const A &other);      // 拷贝构造函数
    A & operate =( const A &other);   // 赋值函数
};

class B
{
      public:
            B(const A &a); // B 的构造函数
      private:
            A  m_a;         // 成员对象
};
```

示例 13-2(a)中，类 B 的构造函数在其初始化表里调用了类 A 的拷贝构造函数，从而将成员对象 m_a 初始化。

示例 13-2 (b)中，类 B 的构造函数在函数体内用赋值的方式将成员对象 m_a 初始化。我们看到的只是一条赋值语句，但实际上 B 的构造函数干了两件事：先暗地里创建 m_a 对象（调用了 A 的无参数构造函数），再调用类 A 的赋值函数，将参数 a 赋给 m_a。

```
B::B(const A &a)
 : m_a(a)
{
   …
}
```

示例 13-2(a) 成员对象在初始化表中被初始化

```
B::B(const A &a)
{
    m_a = a;
    …
}
```

示例 13-2(b) 成员对象在函数体内被初始化

对于内部数据类型的数据成员而言，两种初始化方式的效率几乎没有区别，但后者的程序版式似乎更清晰些。若类 F 的声明如下：

```
class F
{
  public:
    F(int x, int y);        // 构造函数
  private:
    int m_x, m_y;
    int m_i, m_j;
}
```

示例 13-2(c)中 F 的构造函数采用了第一种初始化方式，示例 13-2(d)中 F 的构造函数采用了第二种初始化方式。

```
F::F(int x, int y)
 : m_x(x), m_y(y)
{
   m_i = 0;
   m_j = 0;
}
```

示例 13-2(c) 数据成员在初始化表中被初始化

```
F::F(int x, int y)
{
   m_x = x;
   m_y = y;
   m_i = 0;
   m_j = 0;
}
```

示例 13-2(d) 数据成员在函数体内被初始化

13.3 构造和析构的次序

构造从类层次的最根处开始，在每一层中，首先调用基类的构造函数，然后调用成员对象的构造函数。析构则严格按照与构造相反的次序执行，该次序是唯一的，否则编译器将无法自动执行析构过程。

一个有趣的现象是，成员对象初始化的次序完全不受它们在初始化表中次序的影响，只由成员对象在类中声明的次序决定。这是因为类的声明是唯一的，而类的构造函数可以有多个，因此会有多个不同次序的初始化表。如果成员对象按照初始化表的次序进行构造，这将导致析构函数无法得到唯一的逆序。[Eckel, p260-261]

13.4 示例：类 String 的构造函数与析构函数

```
// String 的普通构造函数
String::String(const char *str)
{
    if(str==NULL)
    {
        m_data = new char[1];
```

```
        *m_data = '\0';
    }
    else
    {
        int length = strlen(str);
        m_data = new char[length+1];
        strcpy(m_data, str);
    }
}

// String 的析构函数
String::~String(void)
{
    delete [] m_data;
    // 由于 m_data 是内部数据类型，也可以写成 delete m_data;
}
```

13.5 不要轻视拷贝构造函数与赋值函数

由于并非所有的对象都会使用拷贝构造函数和赋值函数，程序员可能对这两个函数有些轻视。请先记住以下的警告，在阅读正文时就会多心：

◆ 本章开头讲过，如果不主动编写拷贝构造函数和赋值函数，编译器将以“位拷贝”的方式自动生成缺省的函数。倘若类中含有指针变量，那么这两个缺省的函数就隐含了错误。以类 String 的两个对象 a,b 为例，假设 a.m_data 的内容为“hello”，b.m_data 的内容为“world”。

现将 a 赋给 b，缺省赋值函数的“位拷贝”意味着执行 b.m_data = a.m_data。这将造成三个错误：一是 b.m_data 原有的内存没被释放，造成内存泄露；二是 b.m_data 和 a.m_data 指向同一块内存，a 或 b 任何一方变动都会影响另一方；三是在对象被析构时，m_data 被释放了两次。

◆ 拷贝构造函数和赋值函数非常容易混淆，常导致错写、错用。拷贝构造函数是在对象被创建时调用的，而赋值函数只能被已经存在了的对象调用。以下程序中，第三个语句和第四个语句很相似，你分得清楚哪个调用了拷贝构造函数，哪个调用了赋值函数吗？

```
String  a("hello");
String  b("world");
String  c = a; // 调用了拷贝构造函数，最好写成 c(a);
        c = b; // 调用了赋值函数
```

本例中第三个语句的风格较差，宜改写成 String c(a) 以区别于第四个语句。

13.6 示例：类 String 的拷贝构造函数与赋值函数

```
// 拷贝构造函数
String::String(const String &other)
{
    // 允许操作 other 的私有成员 m_data
    int length = strlen(other.m_data);
    m_data = new char[length+1];
    strcpy(m_data, other.m_data);
}

// 赋值函数
String & String::operate =(const String &other)
{
    // (1) 检查自赋值
    if(this == &other)
        return *this;

    // (2) 释放原有的内存资源
    delete [] m_data;

    // （3）分配新的内存资源，并复制内容
    int length = strlen(other.m_data);
    m_data = new char[length+1];
    strcpy(m_data, other.m_data);

    // （4）返回本对象的引用
    return *this;
}
```

类 String 拷贝构造函数与普通构造函数（参见 13.4 节）的区别是：在函数入口处无需与 NULL 进行比较，这是因为“引用”不可能是 NULL，而“指针”可以为 NULL。

类 String 的赋值函数比构造函数复杂得多，分四步实现：

（1）第一步，检查自赋值。你可能会认为多此一举，难道有人会愚蠢到写出 a = a 这样的自赋值语句！的确不会。但是间接的自赋值仍有可能出现，例如

// 内容自赋值
b = a;
…
	// 地址自赋值
	b = &a;
	…

c = b;
…
a = c;
	a = *b;

也许有人会说：“即使出现自赋值，我也可以不理睬，大不了化点时间让对象复制自己而已，反正不会出错！”

他真的说错了。看看第二步的 delete，自杀后还能复制自己吗？所以，如果发现自赋值，应该马上终止函数。注意不要将检查自赋值的 if 语句

```
if(this == &other)
```

错写成为

```
if( *this == other)
```

（2）第二步，用 delete 释放原有的内存资源。如果现在不释放，以后就没机会了，将造成内存泄露。

（3）第三步，分配新的内存资源，并复制字符串。注意函数 strlen 返回的是有效字符串长度，不包含结束符‘\0’。函数 strcpy 则连‘\0’一起复制。

（4）第四步，返回本对象的引用，目的是为了实现象 a = b = c 这样的链式表达。注意不要将 return *this 错写成 return this 。那么能否写成 return other 呢？效果不是一样吗？

不可以！因为我们不知道参数 other 的生命期。有可能 other 是个临时对象，在赋值结束后它马上消失，那么 return other 返回的将是垃圾。

13.7 偷懒的办法处理拷贝构造函数与赋值函数

如果我们实在不想编写拷贝构造函数和赋值函数，又不允许别人使用编译器生成的缺省函数，怎么办？

偷懒的办法是：只需将拷贝构造函数和赋值函数声明为私有函数，不用编写代码。

例如：

```
class A
{ …
  private:
    A(const A &a);              // 私有的拷贝构造函数
    A & operate =(const A &a);// 私有的赋值函数
};
```

如果有人试图编写如下程序：

```
A  b(a);    // 调用了私有的拷贝构造函数
```

```
b = a;      // 调用了私有的赋值函数
```

编译器将指出错误，因为外界不可以操作 A 的私有函数。

13.8 如何在派生类中实现类的基本函数

基类的构造函数、析构函数、赋值函数都不能被派生类继承。如果类之间存在继承关系，在编写上述基本函数时应注意以下事项：

- 派生类的构造函数应在其初始化表里调用基类的构造函数。
- 基类与派生类的析构函数应该为虚（即加 virtual 关键字）。例如

```
#include <iostream.h>
class Base
{
  public:
    virtual ~Base() { cout<< "~Base" << endl ; }
};

class Derived : public Base
{
  public:
    virtual ~Derived() { cout<< "~Derived" << endl ; }
};

void main(void)
{
    Base * pB = new Derived;  // upcast
    delete pB;
}
```

输出结果为：

~Derived

~Base

如果析构函数不为虚，那么输出结果为

~Base

- 在编写派生类的赋值函数时，注意不要忘记对基类的数据成员重新赋值。例如：

```
class Base
{
    public:
      …
```

```
        Base & operate =(const Base &other);  // 类 Base 的赋值函数
    private:
        int  m_i, m_j, m_k;
};

class Derived : public Base
{
    public:
      …
    Derived & operate =(const Derived &other);// 类 Derived 的赋值函数
    private:
    int  m_x, m_y, m_z;
};

Derived & Derived::operate =(const Derived &other)
{
    //（1）检查自赋值
    if(this == &other)
        return *this;

    //（2）对基类的数据成员重新赋值
    Base::operate =(other);    // 因为不能直接操作私有数据成员

    //（3）对派生类的数据成员赋值
    m_x = other.m_x;
    m_y = other.m_y;
    m_z = other.m_z;

    //（4）返回本对象的引用
    return *this;
}
```

13.9 一些心得体会

有些 C++程序设计书籍称构造函数、析构函数和赋值函数是类的“Big-Three”，它们的确是任何类最重要的函数，不容轻视。

也许你认为本章的内容已经够多了，学会了就能平安无事，我不能作这个保证。如果你希望吃透“Big-Three”，请好好阅读参考文献[Cline]、[Meyers]和 [Murry]。