第十二章 C++函数的高级特性

对比于 C 语言的函数，C++增加了重载（overloaded）、内联（inline）、const 和 virtual 四种新机制。其中重载和内联机制既可用于全局函数也可用于类的成员函数，const 与 virtual 机制仅用于类的成员函数。

重载和内联肯定有其好处才会被 C++语言采纳，但是不可以当成免费的午餐而滥用。本章将探究重载和内联的优点与局限性，说明什么情况下应该采用、不该采用以及要警惕错用。

12.1 函数重载的概念

12.1.1 重载的起源

自然语言中，一个词可以有许多不同的含义，即该词被重载了。人们可以通过上下文来判断该词到底是哪种含义。“词的重载”可以使语言更加简练。例如“吃饭”的含义十分广泛，人们没有必要每次非得说清楚具体吃什么不可。别迂腐得象孔乙己，说茴香豆的茴字有四种写法。

在 C++程序中，可以将语义、功能相似的几个函数用同一个名字表示，即函数重载。这样便于记忆，提高了函数的易用性，这是 C++语言采用重载机制的一个理由。例如示例 12-1-1 中的函数 EatBeef,EatFish,EatChicken 可以用同一个函数名 Eat 表示，用不同类型的参数加以区别。

```
void EatBeef(…);        // 可以改为      void Eat(Beef …);
void EatFish(…);        // 可以改为      void Eat(Fish …);
void EatChicken(…);     // 可以改为      void Eat(Chicken …);
```

示例 12-1-1 重载函数 Eat

C++语言采用重载机制的另一个理由是：类的构造函数需要重载机制。因为 C++规定构造函数与类同名（请参见第十二章），构造函数只能有一个名字。如果想用几种不同的方法创建对象该怎么办？别无选择，只能用重载机制来实现。所以类可以有多个同名的构造函数。

12.1.2 重载是如何实现的

几个同名的重载函数仍然是不同的函数，它们是如何区分的呢？我们自然想到函数接口的两个要素：参数与返回值。

如果同名函数的参数不同（包括类型、顺序不同），那么容易区别出它们是不同的函数。

如果同名函数仅仅是返回值类型不同，有时可以区分，有时却不能。例如：

```
void Function(void);
int  Function (void);
```

上述两个函数，第一个没有返回值，第二个的返回值是 int 类型。如果这样调用函数：

```
int  x = Function ();
```

则可以判断出 Function 是第二个函数。问题是在 C++/C 程序中，我们可以忽略函数的返回值。在这种情况下，编译器和程序员都不知道哪个 Function 函数被调用。

所以只能靠参数而不能靠返回值类型的不同来区分重载函数。编译器根据参数为每个重载函数产生不同的内部标识符。例如编译器为示例 12-1-1 中的三个 Eat 函数产生象_eat_beef、_eat_fish、_eat_chicken 之类的内部标识符（不同的编译器可能产生不同风格的内部标识符）。

如果 C++程序要调用已经被编译后的 C 函数，该怎么办？

假设某个 C 函数的声明如下：

```
void foo(int x, int y);
```

该函数被 C 编译器编译后在库中的名字为 _foo，而 C++编译器则会产生像 _foo_int_int 之类的名字用来支持函数重载和类型安全连接。由于编译后的名字不同，C++程序不能直接调用 C 函数。C++提供了一个 C 连接交换指定符号 extern“C”来解决这个问题。例如：

```
extern “C”
{
   void foo(int x, int y);
   … // 其它函数
}
```

或者写成

```
extern “C”
{
   #include “myheader.h”
   … // 其它 C 头文件
}
```

这就告诉 C++编译译器，函数 foo 是个 C 连接，应该到库中找名字 _foo 而不是找 _foo_int_int。C++编译器开发商已经对 C 标准库的头文件作了 extern“C”处理，所以我们可以用＃include 直接引用这些头文件。

注意并不是两个函数的名字相同就能构成重载。全局函数和类的成员函数同名不算重载，因为函数的作用域不同。例如：

```
void Print(…);     // 全局函数
class A
{…
    void Print(…);// 成员函数
}
```

不论两个 Print 函数的参数是否不同，如果类的某个成员函数要调用全局函数 Print，为了与成员函数 Print 区别，全局函数被调用时应加‘::’标志。如

```
::Print(…);    // 表示 Print 是全局函数而非成员函数
```

12.1.3 当心隐式类型转换导致重载函数产生二义性

示例 12-1-3 中，第一个 output 函数的参数是 int 类型，第二个 output 函数的参数是 float 类型。由于数字本身没有类型，将数字当作参数时将自动进行类型转换（称为隐式类型转换）。语句 output(0.5)将产生编译错误，因为编译器不知道该将 0.5 转换成 int 还是 float 类型的参数。隐式类型转换在很多地方可以简化程序的书写，但是也可能留下隐患。

```
# include <iostream.h>
void output( int x);   // 函数声明
void output( float x); // 函数声明

void output( int x)
{
    cout << " output int " << x << endl ;
}

void output( float x)
{
    cout << " output float " << x << endl ;
}

void main(void)
{
    int   x = 1;
    float y = 1.0;
    output(x);          // output int 1
```

```
    output(y);          // output float 1
    output(1);          // output int 1
//  output(0.5);        // error! ambiguous call, 因为自动类型转换
    output(int(0.5));   // output int 0
    output(float(0.5));// output float 0.5
}
```

示例 12-1-3 隐式类型转换导致重载函数产生二义性

12.2 成员函数的重载、覆盖与隐藏

成员函数的重载、覆盖（override）与隐藏很容易混淆，C++程序员必须要搞清楚概念，否则错误将防不胜防。

12.2.1 重载与覆盖

成员函数被重载的特征：

（1）相同的范围（在同一个类中）；

（2）函数名字相同；

（3）参数不同；

（4）virtual 关键字可有可无。

覆盖是指派生类函数覆盖基类函数，特征是：

（1）不同的范围（分别位于派生类与基类）；

（2）函数名字相同；

（3）参数相同；

（4）基类函数必须有 virtual 关键字。

示例 12-2-1 中，函数 Base::f(int)与 Base::f(float)相互重载，而 Base::g(void)被 Derived::g(void)覆盖。

```
#include <iostream.h>
class Base
{
  public:
          void f(int x){ cout << "Base::f(int) " << x << endl; }
          void f(float x){ cout << "Base::f(float) " << x << endl; }
  virtual void g(void){ cout << "Base::g(void)" << endl;}
};
```

```
class Derived : public Base
```

```
{
  public:
  virtual void g(void){ cout << "Derived::g(void)" << endl;}
};
```

```
void main(void)
{
  Derived  d;
  Base *pb = &d;
  pb->f(42);         // Base::f(int) 42
  pb->f(3.14f);      // Base::f(float) 3.14
  pb->g();           // Derived::g(void)
}
```

示例 12-2-1 成员函数的重载和覆盖

12.2.2 令人迷惑的隐藏规则

本来仅仅区别重载与覆盖并不算困难，但是 C++的隐藏规则使问题复杂性陡然增加。这里“隐藏”是指派生类的函数屏蔽了与其同名的基类函数，规则如下：

（1）如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无 virtual 关键字，基类的函数将被隐藏（注意别与重载混淆）。

（2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有 virtual 关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）。

示例程序 12-2-2（a）中：

（1）函数 Derived::f(float)覆盖了 Base::f(float)。

（2）函数 Derived::g(int)隐藏了 Base::g(float)，而不是重载。

（3）函数 Derived::h(float)隐藏了 Base::h(float)，而不是覆盖。

```
#include <iostream.h>
class Base
{
public:
virtual void f(float x){ cout << "Base::f(float) " << x << endl; }
        void g(float x){ cout << "Base::g(float) " << x << endl; }
        void h(float x){ cout << "Base::h(float) " << x << endl; }
};
```

```
class Derived : public Base
{
public:
```

```
    virtual void f(float x){ cout << "Derived::f(float) " << x << endl; }
            void g(int x){ cout << "Derived::g(int) " << x << endl; }
            void h(float x){ cout << "Derived::h(float) " << x << endl; }
    };
```

示例 12-2-2（a）成员函数的重载、覆盖和隐藏

据作者考察，很多 C++程序员没有意识到有“隐藏”这回事。由于认识不够深刻，“隐藏”的发生可谓神出鬼没，常常产生令人迷惑的结果。

示例 12-2-2（b）中，bp 和 dp 指向同一地址，按理说运行结果应该是相同的，可事实并非这样。

```
    void main(void)
    {
      Derived  d;
      Base *pb = &d;
      Derived *pd = &d;
      // Good : behavior depends solely on type of the object
      pb->f(3.14f);// Derived::f(float) 3.14
      pd->f(3.14f);// Derived::f(float) 3.14

      // Bad : behavior depends on type of the pointer
      pb->g(3.14f);// Base::g(float) 3.14
      pd->g(3.14f);// Derived::g(int) 3        (surprise!)

      // Bad : behavior depends on type of the pointer
      pb->h(3.14f);// Base::h(float) 3.14      (surprise!)
      pd->h(3.14f);// Derived::h(float) 3.14
    }
```

示例 12-2-2（b） 重载、覆盖和隐藏的比较

12.2.3 摆脱隐藏

隐藏规则引起了不少麻烦。示例 12-2-3 程序中，语句 pd->f(10)的本意是想调用函数 Base::f(int)，但是 Base::f(int)不幸被 Derived::f(char *)隐藏了。由于数字 10 不能被隐式地转化为字符串，所以在编译时出错。

```
    class Base
    {
      public:
```

```
    void f(int x);
};
```

```
class Derived : public Base
{
  public:
    void f(char *str);
};
```

```
void Test(void)
{
  Derived *pd = new Derived;
  pd->f(10);    // error
}
```

示例 12-2-3 由于隐藏而导致错误

从示例 12-2-3 看来，隐藏规则似乎很愚蠢。但是隐藏规则至少有两个存在的理由：

- 写语句 pd->f(10)的人可能真的想调用 Derived::f(char *)函数，只是他误将参数写错了。有了隐藏规则，编译器就可以明确指出错误，这未必不是好事。否则，编译器会静悄悄地将错就错，程序员将很难发现这个错误，流下祸根。
- 假如类 Derived 有多个基类（多重继承），有时搞不清楚哪些基类定义了函数 f。如果没有隐藏规则，那么 pd->f(10)可能会调用一个出乎意料的基类函数 f。尽管隐藏规则看起来不怎么有道理，但它的确能消灭这些意外。

示例 12-2-3 中，如果语句 pd->f(10)一定要调用函数 Base::f(int)，那么将类 Derived 修改为如下即可。

```
class Derived : public Base
{
  public:
  void f(char *str);
  void f(int x) { Base::f(x); }
};
```

12.3 参数的缺省值

有一些参数的值在每次函数调用时都相同，书写这样的语句会使人厌烦。C++语言采用参数的缺省值使书写变得简洁（在编译时，缺省值由编译器自动插入）。

参数缺省值的使用规则：

- **【规则 12-3-1】**参数缺省值只能出现在函数的声明中，而不能出现在定义体中。

例如：

```
void Foo(int x=0, int y=0);    // 正确，缺省值出现在函数的声明中
```

```
void Foo(int x=0, int y=0)      // 错误，缺省值出现在函数的定义体中
{
…
}
```

为什么会这样？我想是有两个原因：一是函数的实现（定义）本来就与参数是否有缺省值无关，所以没有必要让缺省值出现在函数的定义体中。二是参数的缺省值可能会改动，显然修改函数的声明比修改函数的定义要方便。

- **【规则 12-3-2】**如果函数有多个参数，参数只能从后向前挨个儿缺省，否则将导致函数调用语句怪模怪样。

  正确的示例如下：

  ```
  void Foo(int x, int y=0, int z=0);
  ```

  错误的示例如下：

  ```
  void Foo(int x=0, int y, int z=0);
  ```

要注意，使用参数的缺省值并没有赋予函数新的功能，仅仅是使书写变得简洁一些。它可能会提高函数的易用性，但是也可能会降低函数的可理解性。所以我们只能适当地使用参数的缺省值，要防止使用不当产生负面效果。示例 12-3-2 中，不合理地使用参数的缺省值将导致重载函数 output 产生二义性。

```
#include <iostream.h>
void output( int x);
void output( int x, float y=0.0);
```

```
void output( int x)
{
    cout << " output int " << x << endl ;
}
```

```
void output( int x, float y)
{
    cout << " output int " << x << " and float " << y << endl ;
}
```

```
void main(void)
{
    int x=1;
    float y=0.5;
//  output(x);          // error! ambiguous call
```

```
    output(x,y);        // output int 1 and float 0.5
}
```

示例 12-3-2　参数的缺省值将导致重载函数产生二义性

12.4 运算符重载

12.4.1 概念

在 C++语言中，可以用关键字 operator 加上运算符来表示函数，叫做运算符重载。例如两个复数相加函数：

```
Complex Add(const Complex &a, const Complex &b);
```

可以用运算符重载来表示：

```
Complex operator +(const Complex &a, const Complex &b);
```

运算符与普通函数在调用时的不同之处是：对于普通函数，参数出现在圆括号内；而对于运算符，参数出现在其左、右侧。例如

```
Complex a, b, c;
…
c = Add(a, b); // 用普通函数
c = a + b;     // 用运算符 +
```

如果运算符被重载为全局函数，那么只有一个参数的运算符叫做一元运算符，有两个参数的运算符叫做二元运算符。

如果运算符被重载为类的成员函数，那么一元运算符没有参数，二元运算符只有一个右侧参数，因为对象自己成了左侧参数。

从语法上讲，运算符既可以定义为全局函数，也可以定义为成员函数。文献[Murray，p44-p47]对此问题作了较多的阐述，并总结了表 12-4-1 的规则。

运算符	规则
所有的一元运算符	建议重载为成员函数
= () [] ->	只能重载为成员函数
+= -= /= *= &= |= ~= %= >>= <<=	建议重载为成员函数
所有其它运算符	建议重载为全局函数

表 12-4-1 运算符的重载规则

由于 C++语言支持函数重载，才能将运算符当成函数来用，C 语言就不行。我们要以平常心来对待运算符重载：

（1）不要过分担心自己不会用，它的本质仍然是程序员们熟悉的函数。

（2）不要过分热心地使用，如果它不能使代码变得更加易读易写，那就别用，否则会自

找麻烦。

12.4.2 不能被重载的运算符

在 C++运算符集合中，有一些运算符是不允许被重载的。这种限制是出于安全方面的考虑，可防止错误和混乱。

（1）不能改变 C++内部数据类型（如 int,float 等）的运算符。

（2）不能重载‘.’，因为‘.’在类中对任何成员都有意义，已经成为标准用法。

（3）不能重载目前 C++运算符集合中没有的符号，如#,@,$等。原因有两点，一是难以理解，二是难以确定优先级。

（4）对已经存在的运算符进行重载时，不能改变优先级规则，否则将引起混乱。

12.5 函数内联

12.5.1 用内联取代宏代码

C++ 语言支持函数内联，其目的是为了提高函数的执行效率（速度）。

在 C 程序中，可以用宏代码提高执行效率。宏代码本身不是函数，但使用起来象函数。预处理器用复制宏代码的方式代替函数调用，省去了参数压栈、生成汇编语言的 CALL 调用、返回参数、执行 return 等过程，从而提高了速度。使用宏代码最大的缺点是容易出错，预处理器在复制宏代码时常常产生意想不到的边际效应。例如

```
#define MAX(a, b)        (a) > (b) ? (a) : (b)
```

语句

```
result = MAX(i, j) + 2 ;
```

将被预处理器解释为

```
result = (i) > (j) ? (i) : (j) + 2 ;
```

由于运算符‘+’比运算符‘:’的优先级高，所以上述语句并不等价于期望的

```
result = ( (i) > (j) ? (i) : (j) ) + 2 ;
```

如果把宏代码改写为

```
#define MAX(a, b)        ( (a) > (b) ? (a) : (b) )
```

则可以解决由优先级引起的错误。但是即使使用修改后的宏代码也不是万无一失的，例如语句

```
result = MAX(i++, j);
```

将被预处理器解释为

```
result = (i++) > (j) ? (i++) : (j);
```

对于 C++ 而言，使用宏代码还有另一种缺点：无法操作类的私有数据成员。

让我们看看 C++ 的“函数内联”是如何工作的。对于任何内联函数，编译器在符号表里放入函数的声明（包括名字、参数类型、返回值类型）。如果编译器没有发现内联函数存在错误，那么该函数的代码也被放入符号表里。在调用一个内联函数时，编译器首先检查调用是否正确（进行类型安全检查，或者进行自动类型转换，当然对所有的函数都一样）。如果正确，内联函数的代码就会直接替换函数调用，于是省去了函数调用的开销。

这个过程与预处理有显著的不同，因为预处理器不能进行类型安全检查，或者进行自动类型转换。假如内联函数是成员函数，对象的地址（this）会被放在合适的地方，这也是预处理器办不到的。

C++ 语言的函数内联机制既具备宏代码的效率，又增加了安全性，而且可以自由操作类的数据成员。所以在 C++ 程序中，应该用内联函数取代所有宏代码，“断言 assert”恐怕是唯一的例外。assert 是仅在 Debug 版本起作用的宏，它用于检查“不应该”发生的情况。为了不在程序的 Debug 版本和 Release 版本引起差别，assert 不应该产生任何副作用。如果 assert 是函数，由于函数调用会引起内存、代码的变动，那么将导致 Debug 版本与 Release 版本存在差异。所以 assert 不是函数，而是宏。（参见 10.5 节“使用断言”）

12.5.2 内联函数的编程风格

关键字 inline 必须与函数定义体放在一起才能使函数成为内联，仅将 inline 放在函数声明前面不起任何作用。如下风格的函数 Foo 不能成为内联函数：

```
inline void Foo(int x, int y);      // inline 仅与函数声明放在一起
void Foo(int x, int y)
{
    …
}
```

而如下风格的函数 Foo 则成为内联函数：

```
void Foo(int x, int y);
inline void Foo(int x, int y)  // inline 与函数定义体放在一起
{
    …
}
```

所以说，**inline 是一种“用于实现的关键字”，而不是一种“用于声明的关键字”**。一般地，用户可以阅读函数的声明，但是看不到函数的定义。尽管在大多数教科书中内联函数的声明、定义体前面都加了 inline 关键字，但我认为 inline 不应该出现在函数的声明中。这个细节虽然不会影响函数的功能，但是体现了高质量 C++/C 程序设计风格的一个基本原则：声明与定义不可混为一谈，用户没有必要、也不应该知道函数是否需要内联。

定义在类声明之中的成员函数将自动地成为内联函数，例如

```
class A
{
public:
    void Foo(int x, int y) { … }  // 自动地成为内联函数
};
```

将成员函数的定义体放在类声明之中虽然能带来书写上的方便，但不是一种良好的编程风格，上例应该改成：

```
// 头文件
class A
{
public:
    void Foo(int x, int y);
}
// 定义文件
inline void A::Foo(int x, int y)
{
    …
}
```

12.5.3 慎用内联

内联能提高函数的执行效率，为什么不把所有的函数都定义成内联函数？

如果所有的函数都是内联函数，还用得着“内联”这个关键字吗？

内联是以代码膨胀（复制）为代价，仅仅省去了函数调用的开销，从而提高函数的执行效率。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。以下情况不宜使用内联：

（1）如果函数体内的代码比较长，使用内联将导致内存消耗代价较高。

（2）如果函数体内出现循环，那么执行函数体内代码的时间要比函数调用的开销大。

类的构造函数和析构函数容易让人误解成使用内联更有效。要当心构造函数和析构函数可能会隐藏一些行为，如“偷偷地”执行了基类或成员对象的构造函数和析构函数。所以不要随便地将构造函数和析构函数的定义体放在类声明中。

一个好的编译器将会根据函数的定义体，自动地取消不值得的内联（这进一步说明了 inline 不应该出现在函数的声明中）。

12.6 一些心得体会

C++ 语言中的重载、内联、缺省参数、隐式转换等机制展现了很多优点，但是这些优点的背后都隐藏着一些隐患。正如人们的饮食，少食和暴食都不可取，应当恰到好处。我们要辨证地看待 C++的新机制，应该恰如其分地使用它们。虽然这会使我们编程时多费一些心思，少了一些痛快，但这才是编程的艺术。