C++ Style and Technique FAQ （中文版）

Bjarne Stroustrup 著， 紫云英 译

Q: 这个简单的程序……我如何把它搞定？

A: 常常有人问我一些简单的程序该如何写，这在学期之初时尤甚。一个典型的问题是：如何读入一些数字，做些处理（比如数学运算），然后输出……好吧好吧，这里我给出一个“通用示范程序”：

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;

int main()
{
        vector<double> v;

        double d;
        while(cin>>d) v.push_back(d); // read elements
        if (!cin.eof()) {           // check if input failed
                cerr << "format error\n";
                return 1;           // error return
        }

        cout << "read " << v.size() << " elements\n";

        reverse(v.begin(),v.end());
        cout << "elements in reverse order:\n";
        for (int i = 0; i<v.size(); ++i) cout << v[i] << '\n';

        return 0; // success return
}
```

程序很简单，是吧。这里是对它的一些“观察报告”：

- 这是一个用标准C++写的程序，使用了标准库[译注：标准库主要是将原来的C运行支持库（Standard C Library）、iostream库、STL（Standard Template Library，标准模板库）等标准化而得的] 。标准库提供的功能都位于namespace

std之中，使用标准库所需包含的头文件是不含.h扩展名的。[译注：有些编译器厂商为了兼容性也提供了含.h扩展名的头文件。]

- 如果你在Windows下编译，你需要把编译选项设为“console application”。记住，你的源代码文件的扩展名必须为.cpp，否则编译器可能会把它当作C代码来处理。
- 主函数main()要返回一个整数。[译注：有些编译器也支持void main()的定义，但这是非标准做法]
- 将输入读入标准库提供的vector容器可以保证你不会犯“缓冲区溢出”之类错误——对于初学者来说，硬是要求“把输入读到一个数组之中，不许犯任何‘愚蠢的错误’”似乎有点过份了——如果你真能达到这样的要求，那你也不能算完全的初学者了。如果你不相信我的这个论断，那么请看看我写的《Learning Standard C++ as a New Language》一文。 [译注：CSDN文档区有该文中译。]
- 代码中“ !cin.eof() ”是用来测试输入流的格式的。具体而言，它测试读输入流的循环是否因遇到EOF而终止。如果不是，那说明输入格式不对（不全是数字）。还有细节地方不清楚，可以参看你使用的教材中关于“流状态”的章节。
- Vector是知道它自己的大小的，所以不必自己清点输入了多少元素。
- 这个程序不含任何显式内存管理代码，也不会产生内存泄漏。Vector会自动配置内存，所以用户不必为此烦心。
- 关于如何读入字符串，请参阅后面的“我如何从标准输入中读取string”条目。
- 这个程序以EOF为输入终止的标志。如果你在UNIX上运行这个程序，可以用Ctrl-D输入EOF。但你用的Windows版本可能会含有一个bug（http://support.microsoft.com/support/kb/articles/Q156/2/58.asp?LN=EN-US&SD=gn&FR=0&qry=End of File&rnk=11&src=DHCS_MSPSS_gn_SRCH&SPR=NTW40），导致系统无法识别EOF字符。如果是这样，那么也许下面这个有稍许改动的程序更适合你：这个程序以单词“end”作为输入终结的标志。

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<string>
using namespace std;

int main()
{
        vector<double> v;

        double d;
        while(cin>>d) v.push_back(d); // read elements
        if (!cin.eof()) {           // check if input failed
                cin.clear();                // clear error state
                string s;
                cin >> s;                   // look for terminator str
                if (s != "end") {
                        cerr << "format error\n";
                        return 1;           // error return
                }
        }
```

```
        cout << "read " << v.size() << " elements\n";

        reverse(v.begin(),v.end());
        cout << "elements in reverse order:\n";
        for (int i = 0; i<v.size(); ++i) cout << v[i] << '\n';

        return 0; // success return
    }
```

《The C++ Programming Language》第三版中关于标准库的章节里有更多更详细例子，你可以通过它们学会如何使用标准库来“轻松搞定简单任务”。

Q: 为何我编译一个程序要花那么多时间？

A: 也许是你的编译器有点不太对头——它是不是年纪太大了，或者没有安装正确？也可能你的电脑该进博物馆了……对于这样的问题我可真是爱莫能助了。

不过，也有可能原因在于你的程序——看看你的程序设计还能不能改进？编译器是不是为了顺利产出正确的二进制码而不得不吃进成百个头文件、几万行的源代码？原则上，只要对源码适当优化一下，编译缓慢的问题应该可以解决。如果症结在于你的类库供应商，那么你大概除了“换一家类库供应商”外确实没什么可做的了；但如果问题在于你自己的代码，那么完全可以通过重构（refactoring）来让你的代码更为结构化，从而使源码一旦有更改时需重编译的代码量最小。这样的代码往往是更好的设计：因为它的藕合程度较低，可维护性较佳。

我们来看一个OOP的经典例子：

```
class Shape {
public:             // interface to users of Shapes
        virtual void draw() const;
        virtual void rotate(int degrees);
        // ...
protected:          // common data (for implementers of Shapes)
        Point center;
        Color col;
        // ...
};

class Circle : public Shape {
public:
        void draw() const;
        void rotate(int) { }
        // ...
protected:
        int radius;
        // ...
```

```
};

class Triangle : public Shape {
public:
        void draw() const;
        void rotate(int);
        // ...
protected:
        Point a, b, c;
        // ...
};
```

上述代码展示的设计理念是：让用户通过Shape的公共界面来处理“各种形状”；而Shape的保护成员提供了各继承类（比如Circle，Triangle）共同需要的功能。也就是说：将各种形状（shapes）的公共因素划归到基类Shape中去。这种理念看来很合理，不过我要提请你注意：

- 要确认“哪些功能会被所有的继承类用到，而应在基类中实作”可不是件简单的事。所以，基类的保护成员或许会随着要求的变化而变化，其频度远高于公共界面之可能变化。例如，尽管我们把“center”作为所有形状的一个属性（从而在基类中声明）似乎是天经地义的，但因此而要在基类中时时维护三角形的中心坐标是很麻烦的，还不如只在需要时才计算——这样可以减少开销。
- 和抽象的公共界面不同，保护成员可能会依赖实作细节，而这是Shape类的使用者所不愿见到的。例如，绝大部分使用Shape的代码应该逻辑上和color无关；但只要color的声明在Shape类中出现了，就往往会导致编译器将定义了“该操作系统中颜色表示”的头文件读入、展开、编译。这都需要时间！
- 当基类中保护成员（比如前面说的center，color）的实作有所变化，那么所有使用了Shape类的代码都需要重新编译——哪怕这些代码中只有很少是真正要用到基类中的那个“语义变化了的保护成员”。

所以，在基类中放一些“对于继承类之实作有帮助”的功能或许是出于好意，但实则是麻烦的源泉。用户的要求是多变的，所以实作代码也是多变的。将多变的代码放在许多继承类都要用到的基类之中，那么变化可就不是局部的了，这会造成全局影响的！具体而言就是：基类所倚赖的一个头文件变动了，那么所有继承类所在的文件都需重新编译。

这样分析过后，解决之道就显而易见了：仅仅把基类用作为抽象的公共界面，而将“对继承类有用”的实作功能移出。

```
class Shape {
public:         // interface to users of Shapes
        virtual void draw() const = 0;
        virtual void rotate(int degrees) = 0;
        virtual Point center() const = 0;
        // ...

        // no data
};
```

```
class Circle : public Shape {
public:
        void draw() const;
        void rotate(int) { }
        Point center() const { return center; }
        // ...
protected:
        Point cent;
        Color col;
        int radius;
        // ...
};

class Triangle : public Shape {
public:
        void draw() const;
        void rotate(int);
        Point center() const;
        // ...
protected:
        Color col;
        Point a, b, c;
        // ...
};
```

这样，继承类的变化就被孤立起来了。由变化带来的重编译时间可以极为显著地缩短。

但是，如果确实有一些功能是要被所有继承类（或者仅仅几个继承类）共享的，又不想在每个继承类中重复这些代码，那怎么办？也好办：把这些功能封装成一个类，如果继承类要用到这些功能，就让它再继承这个类：

```
class Shape {
public:         // interface to users of Shapes
        virtual void draw() const = 0;
        virtual void rotate(int degrees) = 0;
        virtual Point center() const = 0;
        // ...

        // no data
};

struct Common {
        Color col;
        // ...
};

class Circle : public Shape, protected Common {
```

```
public:
        void draw() const;
        void rotate(int) { }
        Point center() const { return center; }
        // ...
protected:
        Point cent;
        int radius;
};

class Triangle : public Shape, protected Common {
public:
        void draw() const;
        void rotate(int);
        Point center() const;
        // ...
protected:
        Point a, b, c;
};
```

[译注：这里作者的思路就是孤立变化，减少耦合。从这个例子中读者可以学到一点Refactoring的入门知识 :O)]

Q: 为何空类的大小不是零?

A: 为了确保两个不同对象的地址不同，必须如此。也正因为如此，new返回的指针总是指向不同的单个对象。我们还是来看代码吧：

```
class Empty { };

void f()
{
        Empty a, b;
        if (&a == &b) cout << "impossible: report error to compiler s

        Empty* p1 = new Empty;
        Empty* p2 = new Empty;
        if (p1 == p2) cout << "impossible: report error to compiler s
}
```

另外，C++中有一条有趣的规则——空基类并不需要另外一个字节来表示：

```
struct X : Empty {
        int a;
        // ...
};
```

```
void f(X* p)
{
        void* p1 = p;
        void* p2 = &p->a;
        if (p1 == p2) cout << "nice: good optimizer";
}
```

如果上述代码中p1和p2相等，那么说明编译器作了优化。这样的优化是安全的，而且非常有用。它允许程序员用空类来表示非常简单的概念，而不需为此付出额外的（空间）代价。一些现代编译器提供了这种“虚基类优化”功能。

Q: 为什么我必须把数据放到类的声明之中？

A: 没人强迫你这么做。如果你不希望界面中有数据，那么就不要把它放在定义界面的类中，放到继承类中好了。参看“为何我编译一个程序要花那么多时间”条目。[译注：本FAQ中凡原文为declare/declaration的均译为声明；define/definition均译为定义。两者涵义之基本差别参见后面“‘int* p;’和‘int *p;’到底哪个正确”条目中的译注。通常而言，我们还是将下面的示例代码称为complex类的**定义**，而将单单一行“class complex;”称作**声明**。]

但也有的时候你确实需要把数据放到类声明里面，比如下面的复数类的例子：

```
template<class Scalar> class complex {
public:
        complex() : re(0), im(0) { }
        complex(Scalar r) : re(r), im(0) { }
        complex(Scalar r, Scalar i) : re(r), im(i) { }
        // ...

        complex& operator+=(const complex& a)
                { re+=a.re; im+=a.im; return *this; }
        // ...
private:
        Scalar re, im;
};
```

这个complex（复数）类是被设计成像C++内置类型那样使用的，所以数据表示必须出现在声明之中，以便可以建立真正的本地对象（即在堆栈上分配的对象，而非在堆中分配），这同时也确保了简单操作能被正确内联化。“本地对象”和“内联”这两点很重要，因为这样才可以使我们的复数类达到和内置复数类型的语言相当的效率。

[译注：我觉得Bjarne的这段回答有点“逃避问题”之嫌。我想，提问者的真实意图或许是想知道如何用C++将“界面”与“实作”完全分离。不幸的是，C++语言和类机制本身不提供这种方式。我们都知道，类的“界面”部分往往被定义为公有（一般是一些虚函数）；“实作”部分则往往定义为保护或私有（包括函数和数据）；但无论是

“public”段还是“protected”、“private”段都必须出现在类的声明中，随类声明所在的头文件一起提供。想来这就是“为何数据必须放到类声明中”问题的由来吧。为了解决这个问题，我们有个变通的办法：使用Proxy模式（参见《Design Patterns : Elements of Reusable Object-Oriented Software》一书），我们可以将实作部分在proxy类中声明（称为“对象组合”），而不将proxy类的声明暴露给用户。例如：

```
class Implementer; // forward declaration

class Interface {
public:
        // interface

private:
        Implementer impl;
};
```

在这个例子中，Implementer类就是proxy。在Interface中暴露给用户的只是一个impl对象的“存根”，而无实作内容。Implementer类可以如下声明：

```
class Implementer {
public:
        // implementation details, including data members

};
```

上述代码中的注释处可以存放提问者所说的“数据”，而Implementer的声明代码不需暴露给用户。不过，Proxy模式也不是十全十美的——Interface通过impl指针间接调用实作代码带来了额外的开销。或许读者会说，C++不是有内联机制吗？这个开销能通过内联定义而弥补吧。但别忘了，此处运用Proxy模式的目的就是把“实作”部分隐藏起来，这“隐藏”往往就意味着“实作代码”以链接库中的二进制代码形式存在。目前的C++编译器和链接器能做到既“代码内联”又“二进制隐藏”吗？或许可以。那么Proxy模式又能否和C++的模板机制“合作愉快”呢？（换句话说，如果前面代码中Interface和Implementer的声明均不是class，而是template，又如何呢？）关键在于，编译器对内联和模板的支持之实作是否需要进行源码拷贝，还是可以进行二进制码拷贝。目前而言，C#的泛型支持之实作是在Intermediate Language层面上的，而C++则是源码层面上的。Bjarne给出的复数类声明代码称“数据必须出现在类声明中”也是部分出于这种考虑。呵呵，扯远了……毕竟，这段文字只是FAQ的“译注”而已，此处不作更多探讨，有兴趣的读者可以自己去寻找答案 :O)]

Q: 为何成员函数不是默认为虚？

A: 因为许多类不是被用来做基类的。[译注：用来做基类的类常类似于其它语言中的interface概念——它们的作用是为一组类定义一个公共介面。但C++中的类显然还有许多其他用途——比如表示一个具体的扩展类型。] 例如，复数类就是如此。

另外，有虚函数的类有虚机制的开销[译注：指存放vtable带来的空间开销和通过vtable

中的指针间接调用带来的时间开销]，通常而言每个对象增加的空间开销是一个字长。这个开销可不小，而且会造成和其他语言（比如C，Fortran）的不兼容性——有虚函数的类的内存数据布局和普通的类是很不一样的。[译注：这种内存数据布局的兼容性问题会给多语言混合编程带来麻烦。]

《The Design and Evolution of C++》 中有更多关于设计理念的细节。

Q: 为何析构函数不是默认为虚？

A: 哈，你大概知道我要说什么了 :O) 仍然是因为——许多类不是被用来做基类的。只有在类被作为interface使用时虚函数才有意义。（这样的类常常在内存堆上实例化对象并通过指针或引用访问。）

那么，何时我该让析构函数为虚呢？哦，答案是——当类有其它虚函数的时候，你就应该让析构函数为虚。有其它虚函数，就意味着这个类要被继承，就意味着它有点“interface”的味道了。这样一来，程序员就可能会以基类指针来指向由它的继承类所实例化而来的对象，而能否通过基类指针来正常释放这样的对象就要看析构函数是否为虚了。 例如：

```
class Base {
        // ...
        virtual ~Base();
};

class Derived : public Base {
        // ...
        ~Derived();
};

void f()
{
        Base* p = new Derived;
        delete p;       // virtual destructor used to ensure that ~Der
}
```

如果Base的析构函数不是虚的，那么Derived的析构函数就不会被调用——这常常会带来恶果：比如，Derived中分配的资源没有被释放。

Q: C++中为何没有虚拟构造函数？

A: 虚拟机制的设计目的是使程序员在不完全了解细节（比如只知该类实现了某个界面，而不知该类确切是什么东东）的情况下也能使用对象。但是，要建立一个对象，可不能只知道“这大体上是什么”就完事——你必须完全了解全部细节，清楚地知道你要建立的对象是究竟什么。所以，构造函数当然不能是虚的了。

不过有时在建立对象时也需要一定的间接性，这就需要用点技巧来实现了。（详见《The C++ Programming Language》，第三版，15.6.2）这样的技巧有时也被称作“虚拟构造函数”。我这里举个使用抽象类来“虚拟构造对象”的例子：

```
struct F {        // interface to object creation functions
        virtual A* make_an_A() const = 0;
        virtual B* make_a_B() const = 0;
};

void user(const F& fac)
{
        A* p = fac.make_an_A();       // make an A of the appropria
        B* q = fac.make_a_B();        // make a B of the appropriat
        // ...
}

struct FX : F {
        A* make_an_A() const { return new AX();      } // AX is deri
        B* make_a_B() const { return new BX();       } // BX is deri

};

struct FY : F {
        A* make_an_A() const { return new AY();      } // AY is deri
        B* make_a_B() const { return new BY();       } // BY is deri

};

int main()
{
        user(FX());     // this user makes AXs and BXs
        user(FY());     // this user makes AYs and BYs
        // ...
}
```

看明白了没有？上述代码其实运用了Factory模式的一个变体。关键之处是，user()被完全孤立开了——它对AX，AY这些类一无所知。（嘿嘿，有时无知有无知的好处 ^_^）

Q: 为何无法在派生类中重载？

A: 这个问题常常是由这样的例子中产生的：

```
#include<iostream>
using namespace std;
```

```
class B {
public:
        int f(int i) { cout << "f(int): "; return i+1; }
        // ...
};

class D : public B {
public:
        double f(double d) { cout << "f(double): "; return d+1.3; }
        // ...
};

int main()
{
        D* pd = new D;

        cout << pd->f(2) << '\n';
        cout << pd->f(2.3) << '\n';
}
```

程序运行结果是：

```
f(double): 3.3
f(double): 3.6
```

而不是某些人（错误地）猜想的那样：

```
f(int): 3
f(double): 3.6
```

换句话说，在D和B之间没有重载发生。你调用了pd->f()，编译器就在D的名字域里找啊找，找到double f(double)后就调用它了。编译器懒得再到B的名字域里去看看有没有哪个函数更符合要求。记住，在C++中，没有跨域重载——继承类和基类虽然关系很亲密，但也不能坏了这条规矩。详见《The Design and Evolution of C++》或者《The C++ Programming Language》第三版。

不过，如果你非得要跨域重载，也不是没有变通的方法——你就把那些函数弄到同一个域里来好了。使用一个using声明就可以搞定。

```
class D : public B {
public:
        using B::f;       // make every f from B available
        double f(double d) { cout << "f(double): "; return d+1.3; }
        // ...
};
```

这样一来，结果就是

```
f(int): 3
f(double): 3.6
```

重载发生了——因为D中的那句 using B::f 明确告诉编译器，要把B域中的f引入当前域，请编译器“一视同仁”。

Q: 我能从构造函数调用虚函数吗？

A: 可以。不过你得悠着点。当你这样做时，也许你自己都不知道自己在干什么！在构造函数中，虚拟机制尚未发生作用，因为此时overriding尚未发生。万丈高楼平地起，总得先打地基吧？对象的建立也是这样——先把基类构造完毕，然后在此基础上构造派生类。

看看这个例子：

```
#include<string>
#include<iostream>
using namespace std;

class B {
public:
        B(const string& ss) { cout << "B constructor\n"; f(ss); }
        virtual void f(const string&) { cout << "B::f\n";}
};

class D : public B {
public:
        D(const string & ss) :B(ss) { cout << "D constructor\n";}
        void f(const string& ss) { cout << "D::f\n"; s = ss; }
private:
        string s;
};

int main()
{
        D d("Hello");
}
```

这段程序经编译运行，得到这样的结果：

```
B constructor
B::f
D constructor
```

注意，输出**不是D::f** 。 究竟发生了什么？f()是在B::B()中调用的。如果构造函数中调用

析构则正相反，遵循从继承类到基类的顺序（拆房子总得从上往下拆吧？），所以其调用虚函数的行为和在构造函数中一样：虚函数**此时此刻**被绑定到哪里（当然应该是基类啦——因为继承类已经被“拆”了——析构了！），调用的就是哪个函数。

更多细节请见《The Design and Evolution of C++》，13.2.4.2 或者《The C++ Programming Language》第三版，15.4.3 。

有时，这条规则被解释为是由于编译器的实作造成的。[译注：从实作角度可以这样解释：在许多编译器中，直到构造函数调用完毕，vtable才被建立，此时虚函数才被动态绑定至继承类的同名函数。] 但事实上不是这么一回事——让编译器实作成“构造函数中调用虚函数也和从其他函数中调用一样”是很简单的[译注：只要把vtable的建立移至构造函数调用之前即可]。关键还在于语言设计时的考量——让虚函数可以求助于基类提供的通用代码。[译注：先有鸡还是先有蛋？Bjarne实际上是在告诉你，不是“先有实作再有规则”，而是“如此实作，因为规则如此”。]

Q: 有"placement delete"吗?

A: 没有。不过如果你真的想要，你就说嘛——哦不，我的意思是——你可以自己写一个。

我们来看看将对象放至某个指定场所的placement new：

```
class Arena {
public:
        void* allocate(size_t);
        void deallocate(void*);

        // ...
};


void* operator new(size_t sz, Arena& a)
{
        return a.allocate(sz);
}


Arena a1(some arguments);
Arena a2(some arguments);
```

现在我们可以写：

```
X* p1 = new(a1) X;
Y* p2 = new(a1) Y;
Z* p3 = new(a2) Z;
// ...
```

但之后我们如何正确删除这些对象？没有内置“placement delete”的理由是，没办法提供一个通用的placement delete。C++的类型系统没办法让我们推断出p1是指向被放置在a1中的对象。即使我们能够非常天才地推知这点，一个简单的指针赋值操作也会让我们重陷茫然。不过，程序员本人应该知道在他自己的程序中什么指向什么，所以可以有解决方案：

```
template<class T> void destroy(T* p, Arena& a)
{
        if (p) {
                p->~T();                // explicit destructor call
                a.deallocate(p);
        }
}
```

这样我们就可以写：

```
destroy(p1,a1);
destroy(p2,a2);
destroy(p3,a3);
```

如果Arena自身跟踪放置其中的对象，那么你可以安全地写出destroy()函数 ，把“保证无错”的监控任务交给Arena，而不是自己承担。

如何在类继承体系中定义配对的operator new() 和 operator delete() 可以参看 《The C++ Programming Language》，Special Edition，15.6节 ，《The Design and Evolution of C++》，10.4节，以及《The C++ Programming Language》，Special Edition，19.4.5节。[译注：此处按原文照译。前面有提到“参见《The C++ Programming Language》第三版”的，实际上特别版（Special Edition）和较近重印的第三版没什么区别。]

Q: 我能防止别人从我的类继承吗？

A: 可以的，但何必呢？好吧，也许有两个理由：

- 出于效率考虑——不希望我的函数调用是虚的
- 出于安全考虑——确保我的类不被用作基类（这样我拷贝对象时就不用担心对象被切割(slicing)了）[译注：“对象切割”指，将派生类对象赋给基类变量时，根据C++的类型转换机制，只有包括在派生类中的基类部分被拷贝，其余部分被“切割”掉了。]

根据我的经验，“效率考虑”常常纯属多余。在C++中，虚函数调用如此之快，和普通函数调用并没有太多的区别。请注意，只有通过指针或者引用调用时才会启用虚拟机制；如果你指名道姓地调用一个对象，C++编译器会自动优化，去除任何的额外开销。

如果为了和“虚函数调用”说byebye，那么确实有给类继承体系“封顶”的需要。在设计前，不访先问问自己，这些函数为何要被设计成虚的。我确实见过这样的例子：性能

要求苛刻的函数被设计成虚的，仅仅因为“我们习惯这样做”！

好了，无论如何，说了那么多，毕竟你只是想知道，为了某种合理的理由，你能不能防止别人继承你的类。答案是可以的。可惜，这里给出的解决之道不够干净利落。你不得不在在你的“封顶类”中虚拟继承一个无法构造的辅助基类。还是让例子来告诉我们一切吧：

```
class Usable;

class Usable_lock {
friend class Usable;
private:
        Usable_lock() {}
        Usable_lock(const Usable_lock&) {}
};

class Usable : public virtual Usable_lock {
// ...
public:
        Usable();
        Usable(char*);
        // ...
};

Usable a;

class DD : public Usable { };

DD dd;  // error: DD::DD() cannot access
         // Usable_lock::Usable_lock(): private member
```

(参见《The Design and Evolution of C++》，11.4.3节)

Q: 为什么我无法限制模板的参数？

A: 呃，其实你是可以的。而且这种做法并不难，也不需要什么超出常规的技巧。

让我们来看这段代码：

```
template<class Container>
void draw_all(Container& c)
{
        for_each(c.begin(),c.end(),mem_fun(&Shape::draw));
}
```

如果c不符合constraints，出现了类型错误，那么错误将发生在相当复杂的for_each解

析之中。比如说，参数化的类型被要求实例化int型，那么我们无法为之调用Shape::draw()。而我们从编译器中得到的错误信息是含糊而令人迷惑的——因为它和标准库中复杂的for_each纠缠不清。

为了早点捕捉到这个错误，我们可以这样写代码：

```
template<class Container>
void draw_all(Container& c)
{
        Shape* p = c.front(); // accept only containers of Shape*s

        for_each(c.begin(),c.end(),mem_fun(&Shape::draw));
}
```

我们注意到，前面加了一行Shape *p的定义（尽管就程序本身而言，p是无用的）。如果不可将c.front()赋给Shape *p，那么就大多数现代编译器而言，我们都可以得到一条含义清晰的出错信息。这样的技巧在所有语言中都很常见，而且对于所有“不同寻常的构造”都不得不如此。[译注：意指对于任何语言，当我们开始探及极限，那么不得不写一些高度技巧性的代码。]

不过这样做不是最好。如果要我来写实际代码，我也许会这样写：

```
template<class Container>
void draw_all(Container& c)
{
        typedef typename Container::value_type T;
        Can_copy<T,Shape*>(); // accept containers of only Shape*

        for_each(c.begin(),c.end(),mem_fun(&Shape::draw));
}
```

这就使代码通用且明显地体现出我的意图——我在使用断言[译注：即明确断言typename Container是draw_all()所接受的容器类型，而不是令人迷惑地定义了一个Shape *指针，也不知道会不会在后面哪里用到]。Can_copy()模板可被这样定义：

```
template<class T1, class T2> struct Can_copy {
        static void constraints(T1 a, T2 b) { T2 c = a; b = a; }
        Can_copy() { void(*p)(T1,T2) = constraints; }
};
```

Can_copy在编译期间检查确认T1可被赋于T2。Can_copy<T,Shape*>检查确认T是一个Shape*类型，或者是一个指向Shape的公有继承类的指针，或者是用户自定义的可被转型为Shape *的类型。注意，这里Can_copy()的实现已经基本上是最优化的了：一行代码用来指明需要检查的constraints[译注：指第1行代码；constraints为T2]，和要对其做这个检查的类型[译注：要作检查的类型为T1]；一行代码用来精确列出所要检查是否满足的constraints（constraints()函数） [译注：第2行之所以要有2个子句并不是重复，而是有原因的。如果T1，T2均是用户自定义的类，那么T2 c = a; 检测能否缺

省构造；b = a; 检测能否拷贝构造]；一行代码用来提供执行这些检查的机会 [译注：指第3行。Can_copy是一个模板类；constraints是其成员函数，第2行只是定义，而未执行]。

[译注：这里constraints实现的关键是依赖C++强大的类型系统，特别是类的多态机制。第2行代码中T2 c = a; b = a; 能够正常通过编译的条件是：T1实现了T2的接口。具体而言，可能是以下4种情况：(1) T1，T2 同类型 (2) 重载operator = (3) 提供了cast operator （类型转换运算符）(4) 派生类对象赋给基类指针。说到这里，记起我曾在以前的一篇文章中说到，C++的genericity实作——template不支持constrained genericity，而Eiffel则从语法级别支持constrained genericity（即提供类似于template <typename T **as Comparable**> xxx 这样的语法——其中Comparable即为一个constraint）。曾有读者指出我这样说是错误的，认为C++ template也支持constrained genericity。现在这部分译文给出了通过使用一些技巧，将OOP和GP的方法结合，从而在C++中巧妙实现constrained genericity的方法。对于爱好C++的读者，这种技巧是值得细细品味的。不过也不要因为太执著于各种细枝末节的代码技巧而丧失了全局眼光。有时语言支持方面的欠缺可以在设计层面（而非代码层面）更优雅地弥补。另外，这能不能算“C++的template支持constrained genericity”，我保留意见。正如，用C通过一些技巧也可以OOP，但我们不说C语言支持OOP。]

请大家再注意，现在我们的定义具备了这些我们需要的特性：

- 你可以不通过定义/拷贝变量就表达出constraints[译注：实则定义/拷贝变量的工作被封装在Can_copy模板中了]，从而可以不必作任何“那个类型是这样被初始化”之类假设，也不用去管对象能否被拷贝、销毁（除非这正是constraints所在）。[译注：即——除非constraints正是“可拷贝”、“可销毁”。如果用易理解的伪码描述，就是template <typename T as Copy_Enabled> xxx，template <typename T as Destructible> xxx 。]
- 如果使用现代编译器，constraints不会带来任何额外代码
- 定义或者使用constraints均不需使用宏定义
- 如果constraints没有被满足，编译器给出的错误消息是容易理解的。事实上，给出的错误消息包括了单词“constraints” (这样，编码者就能从中得到提示)、constraints的名称、具体的出错原因（比如“cannot initialize Shape* by double*”）

既然如此，我们干吗不干脆在C++语言本身中定义类似Can_copy()或者更优雅简洁的语法呢？The Design and Evolution of C++分析了此做法带来的困难。已经有许许多多设计理念浮出水面，只为了让含constraints的模板类易于撰写，同时还要让编译器在constraints不被满足时给出容易理解的出错消息。比方说，我在Can_copy中“使用函数指针”的设计就来自于Alex Stepanov和Jeremy Siek。我认为我的Can_copy()实作还不到可以标准化的程度——它需要更多实践的检验。另外，C++使用者会遭遇许多不同类型的constraints，目前看来还没有哪种形式的带constraints的模板获得压倒多数的支持。

已有不少关于constraints的“内置语言支持”方案被提议和实作。但其实要表述constraint根本不需要什么异乎寻常的东西：毕竟，当我们写一个模板时，我们拥有C++带给我们的强有力的表达能力。让代码来为我的话作证吧：

```
template<class T, class B> struct Derived_from {
        static void constraints(T* p) { B* pb = p; }
        Derived_from() { void(*p)(T*) = constraints; }
};

template<class T1, class T2> struct Can_copy {
        static void constraints(T1 a, T2 b) { T2 c = a; b = a; }
        Can_copy() { void(*p)(T1,T2) = constraints; }
};

template<class T1, class T2 = T1> struct Can_compare {
        static void constraints(T1 a, T2 b) { a==b; a!=b; a<b; }
        Can_compare() { void(*p)(T1,T2) = constraints; }
};

template<class T1, class T2, class T3 = T1> struct Can_multiply {
        static void constraints(T1 a, T2 b, T3 c) { c = a*b; }
        Can_multiply() { void(*p)(T1,T2,T3) = constraints; }
};

struct B { };
struct D : B { };
struct DD : D { };
struct X { };

int main()
{
        Derived_from<D,B>();
        Derived_from<DD,B>();
        Derived_from<X,B>();
        Derived_from<int,B>();
        Derived_from<X,int>();

        Can_compare<int,float>();
        Can_compare<X,B>();
        Can_multiply<int,float>();
        Can_multiply<int,float,double>();
        Can_multiply<B,X>();

        Can_copy<D*,B*>();
        Can_copy<D,B*>();
        Can_copy<int,B*>();
}

// the classical "elements must derived from Mybase*" constraint:

template<class T> class Container : Derived_from<T,Mybase> {
        // ...
```

```
};
```

事实上Derived_from并不检查继承性，而是检查可转换性。不过Derive_from常常是一个更好的名字——有时给constraints起个好名字也是件需细细考量的活儿。

Q: 我们已经有了 "美好的老qsort()"，为什么还要用sort()?

A: 对于初学者而言，

```
qsort(array,asize,sizeof(elem),elem_compare);
```

看上去有点古怪。还是

```
sort(vec.begin(),vec.end());
```

比较好理解，是吧。那么，这点理由就足够让你舍qsort而追求sort了。对于老手来说，sort()要比qsort()快的事实也会让你心动不已。而且sort是泛型的，可以用于任何合理的容器组合、元素类型和比较算法。例如：

```
struct Record {
        string name;
        // ...
};

struct name_compare {          // compare Records using "name" as t
        bool operator()(const Record& a, const Record& b) const
                { return a.name<b.name; }
};

void f(vector<Record>& vs)
{
        sort(vs.begin(), vs.end(), name_compare());
        // ...
}
```

另外，还有许多人欣赏sort()的类型安全性——要使用它可不需要任何强制的类型转换。对于标准类型，也不必写compare()函数，省事不少。如果想看更详尽的解释，参看我的《Learning Standard C++ as a New Language》一文。

另外，为何sort()要比qsort()快？因为它更好地利用了C++的内联语法语义。

Q: 什么是function object?

A: Function object是一个对象，不过它的行为表现像函数。一般而言，它是由一个重载了operator()的类所实例化得来的对象。

Function object的涵义比通常意义上的函数更广泛，因为它可以在多次调用之间保持某种“状态”——这和静态局部变量有异曲同工之妙；不过这种“状态”还可以被初始化，还可以从外面来检测，这可要比静态局部变量强了。我们来看一个例子：

```
class Sum {
        int val;
public:
        Sum(int i) :val(i) { }
        operator int() const { return val; }                    // extract valu

        int operator()(int i) { return val+=i; }  // application
};

void f(vector v)
{
        Sum s = 0;      // initial value 0
        s = for_each(v.begin(), v.end(), s);     // gather the sum of a
        cout << "the sum is " << s << "\n";

        // or even:
        cout << "the sum is " << for_each(v.begin(), v.end(), Sum(0
}
```

这里我要提请大家注意：一个function object可被漂亮地内联化(inlining)，因为对于编译器而言，没有讨厌的指针来混淆视听，所以这样的优化很容易进行。[译注：这指的是将operator()定义为内联函数，可以带来效率的提高。] 作为对比，编译器几乎不可能通过优化将“通过函数指针调用函数”这一步骤所花的开销省掉，至少目前如此。

在标准库中function objects被广泛使用，这给标准库带来了极大的灵活性和可扩展性。

[译注：C++是一个博采众长的语言，function object的概念就是从functional programming中借来的；而C++本身的强大和表现力的丰富也使这种“拿来主义”成为可能。一般而言，在使用function object的地方也常可以使用函数指针；在我们还不熟悉function object的时候我们也常常是使用指针的。但定义一个函数指针的语法可不是太简单明了，而且在C++中指针早已背上了“错误之源”的恶名。更何况，通过指针调用函数增加了间接开销。所以，无论为了语法的优美还是效率的提高，都应该提倡使用function objects。

下面我们再从设计模式的角度来更深入地理解function objects：这是Visitor模式的典型应用。当我们要对某个/某些对象施加某种操作，但又不想将这种操作限定死，那么就可以采用Visitor模式。在Design Patterns一书中，作者把这种模式实作为：通过一个Visitor类来提供这种操作（在前面Bjarne Stroustrup的代码中，Sum就是一个Visitor的变体），用Visitor类实例化一个visitor对象（当然，在前面的代码中对应的是s）；然

后在Iterator的迭代过程中，为每一个对象调用visitor.visit()。这里visit()是Visitor类的一个成员函数，作用相当于Sum类中那个“特殊的成员函数”——operator()；visit()也完全可以被定义为内联函数，以去除间接性，提高性能。在此提请读者注意，C++把重载的操作符也看作函数，只不过是具有特殊函数名的函数。所以实际上Design Patterns一书中Visitor模式的示范实作和这里function object的实作大体上是等价的。一个function object也就是一个特殊的Visitor。]

Q: 我应该怎样处理内存泄漏？

A: 很简单，只要写“不漏”的代码就完事了啊。显然，如果你的代码到处是new、delete、指针运算，那你想让它“不漏”都难。不管你有多么小心谨慎，君为人，非神也，错误在所难免。最终你会被自己越来越复杂的代码逼疯的——你将投身于与内存泄漏的奋斗之中，对bug们不离不弃，直至山峰没有棱角，地球不再转动。而能让你避免这样困境的技巧也不复杂：你只要倚重隐含在幕后的分配机制——构造和析构，让C++的强大的类系统来助你一臂之力就OK了。标准库中的那些容器就是很好的实例。它们让你不必化费大量的时间精力也能轻松惬意地管理内存。我们来看看下面的示例代码——设想一下，如果没有了string和vector，世界将会怎样？如果不用它们，你能第一次就写出毫无内存错误的同样功能代码吗？

```
#include<vector>
#include<string>
#include<iostream>
#include<algorithm>
using namespace std;

int main()      // small program messing around with strings
{
        cout << "enter some whitespace-separated words:\n";
        vector<string> v;
        string s;
        while (cin>>s) v.push_back(s);

        sort(v.begin(),v.end());

        string cat;
        typedef vector<string>::const_iterator Iter;
        for (Iter p = v.begin(); p!=v.end(); ++p) cat += *p+"+";
        cout << cat << '\n';
}
```

请注意这里没有显式的内存管理代码。没有宏，没有类型转换，没有溢出检测，没有强制的大小限制，也没有指针。如果使用function object和标准算法[译注：指标准库中提供的泛型算法]，我连Iterator也可以不用。不过这毕竟只是一个小程序，杀鸡焉用牛刀？

当然，这些方法也并非无懈可击，而且说起来容易做起来难，要系统地使用它们也并不

总是很简单。不过，无论如何，它们的广泛适用性令人惊讶，而且通过移去大量的显式内存分配/释放代码，它们确实增强了代码的可读性和可管理性。早在1981年，我就指出通过大幅度减少需要显式加以管理的对象数量，使用C++“将事情做对”将不再是一件极其费神的艰巨任务。

如果你的应用领域没有能在内存管理方面助你一臂之力的类库，那么如果你还想让你的软件开发变得既快捷又能轻松得到正确结果，最好是先建立这样一个库。

如果你无法让内存分配和释放成为对象的“自然行为”，那么至少你可以通过使用资源句柄来尽量避免内存泄漏。这里是一个示例：假设你需要从函数返回一个对象，这个对象是在自由内存堆上分配的；你可能会忘记释放那个对象——毕竟我们无法通过检查指针来确定其指向的对象是否需要被释放，我们也无法得知谁应该负责释放它。那么，就用资源句柄吧。比如，标准库中的auto_ptr就可以帮助澄清：“释放对象”责任究竟在谁。我们来看：

```
#include<memory>
#include<iostream>
using namespace std;

struct S {
        S() { cout << "make an S\n"; }
        ~S() { cout << "destroy an S\n"; }
        S(const S&) { cout << "copy initialize an S\n"; }
        S& operator=(const S&) { cout << "copy assign an S\n"; }
};

S* f()
{
        return new S; // who is responsible for deleting this S?
};

auto_ptr<S> g()
{
        return auto_ptr<S>(new S);      // explicitly transfe
}

int main()
{
        cout << "start main\n";
        S* p = f();
        cout << "after f() before g()\n";
//      S* q = g();     // caught by compiler
        auto_ptr<S> q = g();
        cout << "exit main\n";
        // leaks *p
        // implicitly deletes *q
}
```

这里只是内存资源管理的例子；至于其它类型的资源管理，可以如法炮制。

如果在你的开发环境中无法系统地使用这种方法（比方说，你使用了第三方提供的古董代码，或者远古“穴居人”参与了你的项目开发），那么你在开发过程中可千万要记住使用内存防漏检测程序，或者干脆使用垃圾收集器（Garbage Collector）。

Q: 为何捕捉到异常后不能继续执行后面的代码呢?

A: 这个问题，换句话说也就是：为什么C++不提供这样一个原语，能使你处理异常过后返回到异常抛出处继续往下执行？[译注：比如，一个简单的resume语句，用法和已有的return语句类似，只不过必须放在exception handler的最后。]

嗯，从异常处理代码返回到异常抛出处继续执行后面的代码的想法很好[译注：现行异常机制的设计是：当异常被抛出和处理后，从处理代码所在的那个catch块往下执行]，但主要问题在于——exception handler不可能知道为了让后面的代码正常运行，需要做多少清除异常的工作[译注：毕竟，当有异常发生，事情就有点不太对劲了，不是吗；更何况收拾烂摊子永远是件麻烦的事]，所以，如果要让“继续执行”能够正常工作，写throw代码的人和写catch代码的人必须对彼此的代码都很熟悉，而这就带来了复杂的相互依赖关系[译注：既指开发人员之间的“相互依赖”，也指代码间的相互依赖——紧耦合的代码可不是好代码哦 :O)]，会带来很多麻烦的维护问题。

在我设计C++的异常处理机制的时候，我曾认真地考虑过这个问题；在C++标准化的过程中，这个问题也被详细地讨论过。（参见《The Design and Evolution of C++》中关于异常处理的章节）如果你想试试看在抛出异常之前能不能解决问题然后继续往下执行，你可以先调用一个“检查—恢复”函数，然后，如果还是不能解决问题，再把异常抛出。一个这样的例子是new_handler。

Q: 为何C++中没有C中realloc()的对应物?

A: 如果你一定想要的话，你当然可以使用realloc()。不过，realloc() 只和通过malloc()之类C函数分配得到的内存“合作愉快”，在分配的内存中不能有具备用户自定义构造函数的对象。请记住：与某些天真的人们的想象相反，realloc()必要时是会拷贝大块的内存到新分配的连续空间中的。所以，realloc没什么好的 ^_^

在C++中，处理内存重分配的较好办法是使用标准库中的容器，比如vector。[译注：这些容器会自己管理需要的内存，在必要时会“增长尺寸”——进行重分配。]

Q: 我如何使用异常处理?

A: 参见《The C++ Programming Language》14章8.3节，以及附录E。附录E主要阐述如何撰写“exception-safe”代码，这个附录可不是写给初学者看的。一个关键技巧是“资源分配即初始化”——这种技巧通过“类的析构函数”给易造成混乱的“资源管理”带来了“秩序的曙光”。

Q: 我如何从标准输入中读取string?

A: 如果要读以空白结束的单个单词，可以这样：

```
#include<iostream>
#include<string>
using namespace std;

int main()
{
        cout << "Please enter a word:\n";

        string s;
        cin>>s;

        cout << "You entered " << s << '\n';
}
```

请注意，这里没有显式的内存管理代码，也没有限制尺寸而可能会不小心溢出的缓冲区。 [译注：似乎Bjarne常骄傲地宣称这点——因为这是string乃至整个标准库带来的重大好处之一，确实值得自豪；而在老的C语言中，最让程序员抱怨的也是内置字符串类型的缺乏以及由此引起的“操作字符串所需要之复杂内存管理措施”所带来的麻烦。Bjarne一定在得意地想，“哈，我的叫C++的小baby终于长大了，趋向完美了！” :O)]

如果你需要一次读一整行，可以这样：

```
#include<iostream>
#include<string>
using namespace std;

int main()
{
        cout << "Please enter a line:\n";

        string s;
        getline(cin, s);

        cout << "You entered " << s << '\n';
}
```

关于标准库所提供之功能的简介（诸如iostream，stream），参见《The C++ Programming Language》第三版的第三章。如果想看C和C++的输入输出功能使用之具体比较，参看我的《Learning Standard C++ as a New Language》一文。

Q: 为何C++不提供“finally”结构？

A: 因为C++提供了另一种机制，完全可以取代finally，而且这种机制几乎总要比finally工作得更好：就是——“分配资源即初始化”。（见《The C++ Programming Language》14.4节）基本的想法是，用一个局部对象来封装一个资源，这样一来局部对象的析构函数就可以自动释放资源。这样，程序员就不会“忘记释放资源”了。 [译注：因为C++的对象“生命周期”机制替他记住了 :O)] 下面是一个例子：

```
class File_handle {
        FILE* p;
public:
        File_handle(const char* n, const char* a)
                { p = fopen(n,a); if (p==0) throw Open_error(errno)
        File_handle(FILE* pp)
                { p = pp; if (p==0) throw Open_error(errno); }

        ~File_handle() { fclose(p); }

        operator FILE*() { return p; }

        // ...
};

void f(const char* fn)
{
        File_handle f(fn,"rw");  // open fn for reading and writing
        // use file through f
}
```

在一个系统中，每一样资源都需要一个“资源局柄”对象，但我们不必为每一个资源都写一个“finally”语句。在实作的系统中，资源的获取和释放的次数远远多于资源的种类，所以“资源分配即初始化”机制产生的代码要比“finally”机制少。

[译注：Object Pascal，Java，C#等语言都有finally语句块，常用于发生异常时对被分配资源的资源的处理——这意味着有多少次分配资源就有多少finally语句块（少了一个finally就意味着有一些资源分配不是“exception safe”的）；而“资源分配即初始化”机制将原本放在finally块中的代码移到了类的析构函数中。我们只需为每一类资源提供一个封装类即可。需代码量孰多孰少？除非你的系统中每一类资源都只被使用一次——这种情况下代码量是相等的；否则永远是前者多于后者 :O)]

另外，请看看《The C++ Programming Language》附录E中的资源管理例子。

Q: 那个auto_ptr是什么东东啊？为什么没有auto_array？

A: 哦，auto_ptr是一个很简单的资源封装类，是在<memory>头文件中定义的。它使

用“资源分配即初始化”技术来保证资源在发生异常时也能被安全释放（“exception safety”）。一个auto_ptr封装了一个指针，也可以被当作指针来使用。当其生命周期到了尽头，auto_ptr会自动释放指针。例如：

```
#include<memory>
using namespace std;

struct X {
        int m;
        // ..
};

void f()
{
        auto_ptr<X> p(new X);
        X* q = new X;

        p->m++;             // use p just like a pointer
        q->m++;
        // ...

        delete q;
}
```

如果在代码用// ...标注的地方抛出异常，那么p会被正常删除——这个功劳应该记在autc

Auto_ptr是一个轻量级的类，没有引入引用计数机制。如果你把一个auto_ptr（比如，ap1）赋给另一个auto_ptr（比如，ap2），那么ap2将持有实际指针，而ap1将持有零指针。例如：

```
#include<memory>
#include<iostream>
using namespace std;

struct X {
        int m;
        // ..
};

int main()
{
        auto_ptr<X> p(new X);
        auto_ptr<X> q(p);
        cout << "p " << p.get() << " q " << q.get() << "\n";
}
```

运行结果应该是先显示一个零指针，然后才是一个实际指针，就像这样：

```
p 0x0 q 0x378d0
```

auto_ptr::get()返回实际指针。

这里，语义似乎是“转移”，而非“拷贝”，这或许有点令人惊讶。特别要注意的是，不要把auto_ptr作为标准容器的参数——标准容器要求通常的拷贝语义。例如：

```
std::vector<auto_ptr<X> >v;  // error
```

一个auto_ptr只能持有指向单个元素的指针，而不是数组指针：

```
void f(int n)
{
        auto_ptr<X> p(new X[n]);      // error
        // ...
}
```

上述代码会出错，因为析构函数是使用delete而非delete[]来释放指针的，所以后面的n-1个X没有被释放。

那么，看来我们应该用一个使用delete[]来释放指针的，叫auto_array的类似东东来放数组了？哦，不，不，没有什么auto_array。理由是，不需要有啊——我们完全可以用vector嘛：

```
void f(int n)
{
        vector<X> v(n);
        // ...
}
```

如果在 // ... 部分发生了异常，v的析构函数会被自动调用。

Q: C和C++风格的内存分配/释放可以混用吗？

A: 可以——从你可在一个程序中同时使用malloc()和new的意义上而言。

不可以——从你无法delete一个以malloc()分配而来之对象的意义上而言。你也无法free()或realloc()一个由new分配而来的对象。

C++的new和delete运算符确保构造和析构正常发生，但C风格的malloc()、calloc()、free()和realloc()可不保证这点。而且，没有任何人能向你担保，new/delete和malloc/free所掌控的内存是相互“兼容”的。如果在你的代码中，两种风格混用而没有给你造成麻烦，那我只能说：直到目前为止，你是非常幸运的 :O)

如果你因为思念“美好的老realloc()”（许多人都思念她）而无法割舍整个古老的C内存分配机制（爱屋及乌？），那么考虑使用标准库中的vector吧。例如：

```
// read words from input into a vector of strings:

vector<string> words;
string s;
while (cin>>s && s!=".") words.push_back(s);
```

Vector会按需要自动增长的。

我的《Learning Standard C++ as a New Language》一文中给出了其它例子，可以参考。

Q: 我想从void *转换，为什么必须使用换型符？

A: 在C中，你可以隐式转换，但这是不安全的，例如：

```
#include<stdio.h>

int main()
{
        char i = 0;
        char j = 0;
        char* p = &i;
        void* q = p;
        int* pp = q;    /* unsafe, legal C, not C++ */

        printf("%d %d\n",i,j);
        *pp = -1;       /* overwrite memory starting at &i */
        printf("%d %d\n",i,j);
}
```

如果你使用T*类型的指针，该指针却不指向T类型的对象，后果可能是灾难性的；所以在C++中如果你要将void*换型为T*，你必须使用显式换型：

```
int* pp = (int*)q;
```

或者，更好的是，使用新的换型符，以使换型操作更为醒目：

```
int* pp = static_cast<int*>(q);
```

当然，最好的还是——不要换型。

在C中一类最常见的不安全换型发生在将malloc()分配而来的内存赋给某个指针之时，例如：

```
int* p = malloc(sizeof(int));
```

在C++中，应该使用类型安全的new操作符：

```
int* p = new int;
```

而且，new还有附带的好处：

- new不会“偶然”地分配错误大小的内存
- new自动检查内存是否已枯竭
- new支持初始化

例如：

```
typedef std::complex<double> cmplx;

/* C style: */
cmplx* p = (cmplx*)malloc(sizeof(int));        /* error: wrong size *
                                               /* forgot to test for p=
if (*p == 7) { /* ... */ }                     /* oops: forgot to initi

// C++ style:
cmplx* q = new cmplx(1,2); // will throw bad_alloc if memory is exha
if (*q == 7) { /* ... */ }
```

A: 如何在类中定义常量?

Q: 如果你想得到一个可用于常量表达式中的常量，例如数组大小的定义，那么你有两种选择：

```
class X {
        static const int c1 = 7;
        enum { c2 = 19 };

        char v1[c1];
        char v2[c2];

        // ...
};
```

一眼望去，c1的定义似乎更加直截了当，但别忘了只有static的整型或枚举型量才能如此

```
class Y {
        const int c3 = 7;               // error: not static
        static int c4 = 7;              // error: not const
        static const float c5 = 7;      // error not integral
};
```

我还是更喜欢玩“enum戏法”，因为这种定义可移植性好，而且不会引诱我去使用非标

那么，为何要有这些不方便的限制？因为类通常声明在头文件中，而头文件往往被许多单元所包含。[所以，类可能会被重复声明。]但是，为了避免链接器设计的复杂化，C++要求每个对象都只能被定义一次。如果C++允许类内定义要作为对象被存在内存中的实体，那么这项要求就无法满足了。关于C++设计时的一些折衷，参见《The Design and Evolution of C++》。

如果这个常量不需要被用于常量表达式，那么你的选择余地就比较大了：

```
class Z {
        static char* p;         // initialize in definition
        const int i;            // initialize in constructor
public:
        Z(int ii) :i(ii) { }
};

char* Z::p = "hello, there";
```

只有当static成员是在类外被定义的，你才可以获取它的地址，例如：

```
class AE {
        // ...
public:
        static const int c6 = 7;
        static const int c7 = 31;
};

const int AE::c7;       // definition

int f()
{
        const int* p1 = &AE::c6;        // error: c6 not an lvalue
        const int* p2 = &AE::c7;        // ok
        // ...
}
```

Q: 为何delete操作不把指针置零？

A: 嗯，问得挺有道理的。我们来看：

```
delete p;
// ...
delete p;
```

如果代码中的//...部分没有再次给p分配内存，那么这段代码就对同一片内存释放了两次。这是个严重的错误，可惜C++无法有效地阻止你写这种代码。不过，我们都知道，释放空指针是无危害的，所以如果在每一个delete p;后面都紧接一个p = 0;，那么两次释放同一片内存的错误就不会发生了。尽管如此，在C++中没有任何语法可以强制程序员在释放指针后立刻将该指针归零。所以，看来避免犯这样的错误的重任只能全落在程序员肩上了。或许，delete自动把指针归零真是个好主意？

哦，不不，这个主意不够“好”。一个理由是，被delete的指针未必是左值。我们来看：

```
delete p+1;
delete f(x);
```

你让delete把什么自动置零？也许这样的例子不常见，但足可证明“delete自动把指针归零”并不保险。[译注：事实上，我们真正想要的是：“任何指向被释放的内存区域的指针都被自动归零”——但可惜除了Garbage Collector外没什么东东可以做到这点。]再来看个简单例子：

```
T* p = new T;
T* q = p;
delete p;
delete q;       // ouch!
```

C++标准其实允许编译器实作为“自动把传给delete的左值置零”，我也希望编译器厂商这样做，但看来厂商们并不喜欢这样。一个理由就是上述例子——第3行语句如果delete把p自动置零了又如何呢？q又没有被自动置零，第4行照样出错。

如果你觉得释放内存时把指针置零很重要，那么不妨写这样一个destroy函数：

```
template<class T> inline void destroy(T*& p) { delete p; p = 0; }
```

不妨把delete带来的麻烦看作“尽量少用new/delete，多用标准库中的容器”之另一条理由吧 :O)

请注意，把指针作为引用传递（以便delete可以把指针置零）会带来额外的效益——防止右值被传递给destroy()：

```
int* f();
int* p;
// ...
destroy(f());    // error: trying to pass an rvalue by non-const refere
```

```
destroy(p+1);  // error: trying to pass an rvalue by non-const referer
```

Q: 我可以写"void main()"吗？

A: 这样的定义

```
void main() { /* ... */ }
```

不是C++，也不是C。（参见ISO C++ 标准 3.6.1[2] 或 ISO C 标准 5.1.2.2.1）
一个遵从标准的编译器实作应该接受

```
int main() { /* ... */ }
```

和

```
int main(int argc, char* argv[]) { /* ... */ }
```

编译器也可以提供main()的更多重载版本，不过它们都必须返回int，这个int是返回给你的程序的调用者的，这是种“负责”的做法，“什么都不返回”可不大好哦。如果你程序的调用者不支持用“返回值”来交流，这个值会被自动忽略——但这也不能使void main()成为合法的C++或C代码。即使你的编译器支持这种定义，最好也不要养成这种习惯——否则你可能被其他C/C++认为浅薄无知哦。

在C++中，如果你嫌麻烦，可以不必显式地写出return语句。编译器会自动返回0。例如：

```
#include<iostream>

int main()
{
        std::cout << "This program returns the integer value 0\n";
}
```

麻烦吗？不麻烦，int main()比void main()还少了一个字母呢 :O)另外，还要请你注意：无论是ISO C++还是C99都不允许你省略返回类型定义。这也就是说，和C89及ARM C++[译注：指Margaret Ellis和Bjarne Stroustrup于1990年合著的《The Annotated C++ Reference Manual》中描述的C++]不同，int并不是缺省返回值。所以，

```
#include<iostream>

main() { /* ... */ }
```

会出错，因为main()函数缺少返回类型。

Q: 为何我不能重载“.”、“::”和“sizeof”等操作符？

A: 大部分的操作符是可以被重载的，例外的只有“.”、“::”、“?:”和“sizeof”。没有什么非禁止operator?:重载的理由，只不过没有必要而已。另外，expr1? expr2:expr3的重载函数无法保证expr2和expr3中只有一个被执行。

而“sizeof”无法被重载是因为不少内部操作，比如指针加法，都依赖于它，例如：

```
X a[10];
X* p = &a[3];
X* q = &a[3];
p++;    // p points to a[4]
        // thus the integer value of p must be
        // sizeof(X) larger than the integer value of q
```

这样，sizeof(X)无法在不违背基本语言规则的前提下表达什么新的语义。

在N::m中，N和m都不是表达式，它们只是编译器“认识”的名字，“::”执行的实际操作是编译时的名字域解析，并没有表达式的运算牵涉在内。或许有人会觉得重载一个“x::y”（其中x是实际对象，而非名字域或类名）是一个好主意，但这样做引入了新的语法[译注：重载的本意是让操作符可以有新的语义，而不是更改语法——否则会引起混乱]，我可不认为新语法带来的复杂性会给我们什么好处。

原则上来说，“.”运算符是可以被重载的，就像“->”一样。不过，这会带来语义的混淆——我们到底是想和“.”后面的对象打交道呢，还是“.”后面的东东所实际指向的实体打交道呢？看看这个例子（它假设“.”重载是可以的）：

```
class Y {
public:
        void f();
        // ...
};

class X {       // assume that you can overload .
        Y* p;
        Y& operator.() { return *p; }
        void f();
        // ...
};

void g(X& x)
{
        x.f();  // X::f or Y::f or error?
```

```
}
```

这个问题有好几种解决方案。在C++标准化之时，何种方案为佳并不明显。细节请参见《The Design and Evolution of C++》。

Q: 我怎样才能把整数转化为字符串？

A: 最简单的方法是使用stringstream ：

```
#include<iostream>
#include<string>
#include<sstream>
using namespace std;

string itos(int i) // convert int to string
{
        stringstream s;
        s << i;
        return s.str();
}

int main()
{
        int i = 127;
        string ss = itos(i);
        const char* p = ss.c_str();

        cout << ss << " " << p << "\n";
}
```

当然，很自然地，你可以用这种方法来把任何可通过“<<”输出的类型转化为string。想知道string流的细节吗？参见《The C++ Programming Language》，21.5.3节。

Q: “int* p;”和“int *p;”，到底哪个正确？

A: 如果让计算机来读，两者完全等同，都是正确的。我们还可以声明成“int * p”或“int*p”。编译器不会理会你是不是在哪里多放了几个空格。

不过如果让人来读，两者的含义就有所不同了。代码的书写风格是很重要的。C风格的表达式和声明式常被看作比“necessary evil”[译注：“必要之恶”，意指为了达到某种目的而不得不付出的代价。例如有人认为环境的破坏是经济发展带来的“necessary evil”]更糟的东西，而C++则很强调类型。所以，“int *p”和“int* p”之间并无对错之分，只有风格之争。

一个典型的C程序员会写“int *p”，而且振振有词地告诉你“这表示‘*p是一个int’”——听上去挺有道理的。这里，*和p绑在了一起——这就是C的风格。这种风格强调的是语法。

而一个典型的C++程序员会写“int* p”，并告诉你“p是一个指向int的指针，p的类型是int*”。这种风格强调的是类型。当然，我喜欢这种风格 :O) 而且，我认为，类型是非常重要的概念，我们应该注重类型。它的重要性丝毫不亚于C++语言中的其它“较为高级的部分”。[译注：诸如RTTI，各种cast，template机制等，可称为“较高级的部分”了吧，但它们其实也是类型概念的扩展和运用。我曾写过两篇谈到C++和OOP的文章发表在本刊上，文中都强调了理解“类型”之重要性。我还曾译过Object Unencapsulated（这本书由作者先前所著在网上广为流传的C++?? A Critique修订而来）中讲类型的章节，这本书的作者甚至称Object Oriented Programming应该正名为Type Oriented Programming——“面向类型编程”！这有点矫枉过正了，但类型确是编程语言之核心部分。]

当声明单个变量时，int *和int*的差别并不是特别突出，但当我们要一次声明多个变量时，易混淆之处就全暴露出来了：

```
int* p, p1;       // probable error: p1 is not an int*
```

这里，p1的类型到底是int还是int *呢？把*放得离p近一点也同样不能澄清问题：

```
int *p, p1;       // probable error?
```

看来为了保险起见，只好一次声明一个变量了——特别是当声明伴随着初始化之时。[译注：本FAQ中凡原文为declare/declaration的均译为声明；define/definition均译为定义。通常认为，两者涵义之基本差别是：“声明”只是为编译器提供信息，让编译器在符号表中为被声明的符号（比如类型名，变量名，函数名等）保留位置，而不用指明该符号所对应的具体语义——即：没有任何内存空间的分配或者实际二进制代码的生成。而“定义”则须指明语义——如果把“声明”比作在辞典中为一个新词保留条目；那么“定义”就好比在条目中对这个词的意思、用法给出详细解释。当我们说一个C++语句是“定义”，那么编译器必定会为该语句产生对应的机器指令或者分配内存，而被称为“声明”的语句则不会被编译出任何实际代码。从这个角度而言，原文中有些地方虽作者写的是“对象、类、类型的声明（declaration）”，但或许改译为“定义”较符合我们的理解。不过本译文还是采用忠于原文的译法，并不按照我的理解而加以更改。特此说明。另外，译文中凡涉及我个人对原文的理解、补充之部分均以译注形式给出，供读者参考。]人们一般不太可能写出像这样的代码：

```
int* p = &i;
int p1 = p;       // error: int initialized by int*
```

如果真的有人这样写，编译器也不会同意——它会报错的。

每当达到某种目的有两条以上途径，就会有些人被搞糊涂；每当一些选择是出于个人喜好，争论就会无休无止。坚持一次只声明一个指针并在声明时顺便初始化，困扰我们已久的混淆之源就会随风逝去。如果你想了解有关C的声明语法的更多讨论，参见《The

Design and Evolution of C++》 。

Q: 何种代码布局风格为佳？

A: 哦，这是个人品味问题了。人们常常很重视代码布局之风格，但或许风格的一致性要比选择何种风格更重要。如果非要我为我的个人偏好建立“逻辑证明”，和别人一样，我会头大的 :O)

我个人喜欢使用“K&R”风格，如果算上那些C语言中不存在的构造之使用惯例，那么人们有时也称之为“Stroustrup”风格。例如：

```
class C : public B {
public:
        // ...
};

void f(int* p, int max)
{
        if (p) {
                // ...
        }

        for (int i = 0; i<max; ++i) {
                // ...
        }
}
```

这种风格比较节省“垂直空间”——我喜欢让尽量多的内容可以显示在一屏上 :O) 而函数定义开始的花括号之所以如此放置，是因为这样一来就和类定义区分开来，我就可以一眼看出：噢，这是函数！

正确的缩进非常重要。

一些设计问题，比如使用抽象类来表示重要的界面、使用模板来表示灵活而可扩展的类型安全抽象、正确使用“异常”来表示错误，远远要比代码风格重要。

[译注：《The Practice of Programming》中有一章对“代码风格”问题作了详细的阐述。]

Q: 我该把const写在类型前面还是后面？

A: 我是喜欢写在前面的。不过这只是个人口味的问题。“const T”和“T const”均是允许的，而且它们是等价的。例如：

```
const int a = 1;// ok
int const b = 2;// also ok
```

我想，使用第一种写法更合乎语言习惯，比较不容易让人迷惑 :O)

为什么会这样？当我发明“const”（最早是被命名为“readonly”且有一个叫“writeonly”的对应物）时，我让它在前面和后面都行，因为这不会带来二义性。当时的C/C++编译器对修饰符很少有强加的语序规则。

我不记得当时有过什么关于语序的深思熟虑或相关的争论。一些早期的C++使用者（特别是我）当时只是单纯地觉得const int c = 10;要比int const c = 10;好看而已。或许，我是受了这件事实的影响：许多我早年写的例子是用“readonly”修饰的，而readonly int c = 10;确实看上去要比int readonly c = 10;舒服。而最早的使用“const”的C/C++代码是我用全局查找替换功能把readonly换成const而来的。我还记得和几个人讨论过关于语法“变体”问题，包括Dennis Ritchie。不过我不记得当时我们谈的是哪几种语言了。

另外，请注意：如果指针本身不可被修改，那么const应该放在“*”的后面。例如：

```
int *const p1 = q;      // constant pointer to int variable
int const* p2 = q;      // pointer to constant int
const int* p3 = q;      // pointer to constant int
```

Q: 宏有什么不好吗？

A: 宏不遵循C++的作用域和类型规则，这会带来许多麻烦。因此，C++提供了能和语言其它部分“合作愉快”的替代机制，比如内联函数、模板、名字空间机制。让我们来看这样的代码：

```
#include "someheader.h"
struct S {
        int alpha;
        int beta;
 };
```

如果有人(不明智地)写了一个叫“alpha”或者“beta”的宏，那么这段代码无法通过编

```
#define alpha 'a'
#define beta b[2]
```

那么前面的代码就完全背离本意了。

把宏（而且只有宏）的名称全部用大写字母表示确实有助于缓解问题，但宏是没有语言级保护机制的。例如，在以上例子中alpha和beta在S的作用域中，是S的成员变量，但这对于宏毫无影响。宏的展开是在编译前进行的，展开程序只是把源文件看作字符流而

已。这也是C/C++程序设计环境的欠缺之处：计算机和电脑眼中的源文件的涵义是不同的。

不幸的是，你无法确保其他程序员不犯你所认为的“愚蠢的”错误。比方说，近来有人告诉我，他们遇到一个含“goto”语句的宏。我见到过这样的代码，也听到过这样的论点——有时宏中的“goto”是有用的。例如：

```
#define prefix get_ready(); int ret__
#define Return(i) ret__=i; do_something(); goto exit
#define suffix exit: cleanup(); return ret__

void f()
{
        prefix;
        // ...
        Return(10);
        // ...
        Return(x++);
        //...
        suffix;
}
```

如果你是一个负责维护的程序员，这样的代码被提交到你面前，而宏定义（为了给这个“戏法”增加难度而）被藏到了一个头文件中（这种情况并非罕见），你作何感想？是不是一头雾水？

一个常见而微妙的问题是，函数风格的宏不遵守函数参数调用规则。例如：

```
#define square(x) (x*x)

void f(double d, int i)
{
        square(d);      // fine
        square(i++);    // ouch: means (i++*i++)
        square(d+1);    // ouch: means (d+1*d+1); that is, (d+d+1)
        // ...
}
```

“d+1”的问题可以通过给宏定义加括号解决：

```
#define square(x) ((x)*(x))     /* better */
```

但是，“i++”被执行两次的问题仍然没有解决。

我知道有些（其它语言中）被称为“宏”的东西并不象C/C++预处理器所处理的“宏”那样缺陷多多、麻烦重重，但我并不想改进C++的宏，而是建议你正确使用C++语言中

的其他机制，比如内联函数、模板、构造函数、析构函数、异常处理等。

[译注：以上是Bjarne Stroustrup的C++ Style and Technique FAQ的全文翻译。Bjarne是丹麦人，他写的英文文章可不好读，技术文章尤甚。本译文或许错误偏颇之处不少，欢迎广大读者指正。我的email是 zmelody@sohu.com 。]