

```
int x = {5};  
double y {2.75};  
short quar[5] {4, 5, 2, 76, 1};
```

另外，列表初始化语法也可用于new表达式中：

```
int * ar = new int [4] {2, 4, 6, 7}; // C++11
```

创建对象时，也可使用大括号（而不是圆括号）括起的列表来调用构造函数：

```
class Stump  
{  
private:  
    int roots;  
    double weight;  
public:  
    Stump(int r, double w) : roots(r), weight(w) {}  
};  
Stump s1(3, 15.6); // old style  
Stump s2{5, 43.4}; // C++11  
Stump s3 = {4, 32.1}; // C++11
```

然而，如果类有将模板std::initializer_list作为参数的构造函数，则只有该构造函数可以使用列表初始化形式。第3章、4章、9章、10章和第16章讨论了列表初始化的各个方面。

1. 缩窄

初始化列表语法可防止缩窄，即禁止将数值赋给无法存储它的数值变量。常规初始化允许程序员执行可能没有意义的操作：

```
char c1 = 1.57e27; // double-to-char, undefined behavior  
char c2 = 459585821; // int-to-char, undefined behavior
```

然而，如果使用初始化列表语法，编译器将禁止进行这样的类型转换，即将值存储到比它“窄”的变量中：

```
char c1 {1.57e27};      // double-to-char, compile-time error
char c2 = {459585821}; // int-to-char,out of range, compile-time error
```

但允许转换为更宽的类型。另外，只要值在较窄类型的取值范围内，将其转换为较窄的类型也是允许的：

```
char c1 {66};          // int-to-char, in range, allowed
double c2 = {66}; // int-to-double, allowed
```

2. std::initializer_list

C++11提供了模板类initializer_list，可将其用作构造函数的参数，这在第16章讨论过。如果类有接受initializer_list作为参数的构造函数，则初始化列表语法就只能用于该构造函数。列表中的元素必须是同一种类型或可转换为同一种类型。STL容器提供了将initializer_list作为参数的构造函数：

```
vector<int> a1(10);    // uninitialized vector with 10 elements
vector<int> a2{10};    // initializer-list, a2 has 1 element set to 10
vector<int> a3{4,6,1}; // 3 elements set to 4,6,1
```

头文件initializer_list提供了对模板类initializer_list的支持。这个类包含成员函数begin()和end()，可用于获悉列表的范围。除用于构造函数外，还可将initializer_list用作常规函数的参数：

替换为如下代码：

```
for (auto p = il.begin(); p != il.end(); p++)
```

2. decltype

关键字decltype将变量的类型声明为表达式指定的类型。下面的语句的含义是，让y的类型与x相同，其中x是一个表达式：

```
decltype(x) y;
```

下面是几个示例：

```
double x;
int n;
decltype(x*n) q; // q same type as x*n, i.e., double
decltype(&x) pd; // pd same as &x, i.e., double *
```

这在定义模板时特别有用，因为只有等到模板被实例化时才能确定类型：

```
template<typename T, typename U>
void ef(T t, U u)
{
    decltype(T*U) tu;
    ...
}
```

其中tu将为表达式TU的类型，这里假定定义了运算TU。例如，如果T为char，U为short，则tu将为int，这是由整型算术自动执行整型提升导致的。

decltype的工作原理比auto复杂，根据使用的表达式，指定的类型可以为引用和const。下面是几个示例：

```
int j = 3;
int &k = j
const int &n = j;
decltype(n) i1;           // i1 type const int &
decltype(j) i2;           // i2 type int
decltype((j)) i3;         // i3 type int &
decltype(k + 1) i4;        // i4 type int
```

有关导致上述结果的规则的详细信息，请参阅第8章。

3. 返回类型后置

C++11新增了一种函数声明语法：在函数名和参数列表后面（而不是前面）指定返回类型：

```
double f1(double, int);           // traditional syntax
auto f2(double, int) -> double;  // new syntax, return type is double
```

就常规函数的可读性而言，这种新语法好像是倒退，但让您能够使用`decltype`来指定模板函数的返回类型：

```
template<typename T, typename U>
auto eff(T t, U u) -> decltype(T*U)
{
    ...
}
```

这里解决的问题是，在编译器遇到`eff`的参数列表前，`T`和`U`还不在作用域内，因此必须在参数列表后使用`decltype`。这种新语法使得能够这样做。

4. 模板别名：`using =`

对于冗长或复杂的标识符，如果能够创建其别名将很方便。以前，C++为此提供了typedef：

```
typedef std::vector<std::string>::iterator itType;
```

C++11提供了另一种创建别名的语法，这在第14章讨论过：

```
using itType = std::vector<std::string>::iterator;
```

差别在于，新语法也可用于模板部分具体化，但typedef不能：

```
template<typename T>
using arr12 = std::array<T, 12>; // template for multiple aliases
```

上述语句具体化模板array<T, int>（将参数int设置为12）。例如，对于下述声明：

```
std::array<double, 12> a1;
std::array<std::string, 12> a2;
```

可将它们替换为如下声明：

```
arr12<double> a1;
arr12<std::string> a2;
```

5. nullptr

空指针是不会指向有效数据的指针。以前，C++在源代码中使用0表示这种指针，但内部表示可能不同。这带来了一些问题，因为这使得0即可表示指针常量，又可表示整型常量。正如第12章讨论的，C++11新增了关键字nullptr，用于表示空指针；它是指针类型，不能转换为整型类型。为向后兼容，C++11仍允许使用0来表示空指针，因此表达式nullptr == 0为true，但使用nullptr而不是0提供了更高的类型安全。例如，可将0传递给接受int参数的函数，但如果试图将nullptr传递给这样的函数，编译器将此视为错误。因此，出于清晰和安全考虑，请使用nullptr—如果您的编译器支持它。

18.1.4 智能指针

如果在程序中使用new从堆（自由存储区）分配内存，等到不再需要时，应使用delete将其释放。C++引入了智能指针auto_ptr，以帮助自动完成这个过程。随后的编程体验（尤其是使用STL时）表明，需要有更精致的机制。基于程序员的编程体验和BOOST库提供的解决方案，C++11摒弃了auto_ptr，并新增了三种智能指针：unique_ptr、shared_ptr和weak_ptr，第16章讨论了前两种。

所有新增的智能指针都能与STL容器和移动语义协同工作。

18.1.5 异常规范方面的修改

以前，C++提供了一种语法，可用于指出函数可能引发哪些异常（参见第15章）：

```
void f501(int) throw(bad_dog); // can throw type bad_dog exception
void f733(long long) throw(); // doesn't throw an exception
```

与auto_ptr一样，C++编程社区的集体经验表明，异常规范的效果没有预期的好。因此，C++11摒弃的异常规范。然而，标准委员会认为，指出函数不会引发异常有一定的价值，他们为此添加了关键字noexcept：

```
void f875(short, short) noexcept; // doesn't throw an exception
```

18.1.6 作用域内枚举

传统的C++枚举提供了一种创建名称常量的方式，但其类型检查相当低级。另外，枚举名的作用域为枚举定义所属的作用域，这意味着如果在同一个作用域内定义两个枚举，它们的枚举成员不能同名。最后，枚举可能不是可完全移植的，因为不同的实现可能选择不同的底层类型。为解决这些问题，C++11新增了一种枚举。这种枚举使用class或struct定义：

```
enum Old1 {yes, no, maybe}; // traditional form
enum class New1 {never, sometimes, often, always}; // new form
enum struct New2 {never, lever, sever}; // new form
```

新枚举要求进行显式限定，以免发生名称冲突。因此，引用特定枚举时，需要使用New1::never和New2::never等。更详细的信息请参阅第

10章。

18.1.7 对类的修改

为简化和扩展类设计，C++11做了多项改进。这包括允许构造函数被继承和彼此调用、更佳的方法访问控制方式以及移动构造函数和移动赋值运算符，这些都将在本章介绍。下面先来复习本书前面介绍过的改进。

1. 显式转换运算符

有趣的是，C++很早就支持对象自动转换。但随着编程经验的积累，程序员逐渐认识到，自动类型转换可能导致意外转换的问题。为解决这种问题，C++引入了关键字`explicit`，以禁止单参数构造函数导致的自动转换：

```
class Plebe
{
    Plebe(int);    // automatic int-to-plebe conversion
    explicit Plebe(double); // requires explicit use
    ...
};

...
Plebe a, b;
a = 5;           // implicit conversion, call Plebe(5)
b = 0.5;         // not allowed
b = Plebe(0.5); // explicit conversion
```

C++11拓展了`explicit`的这种用法，使得可对转换函数做类似的处理（参见第11章）：

```

class Plebe
{
    ...
    // conversion functions
    operator int() const;
    explicit operator double() const;
    ...
};

...
Plebe a, b;
int n = a;           // int-to-Plebe automatic conversion
double x = b;        // not allowed
x = double(b);      // explicit conversion, allowed

```

2. 类内成员初始化

很多首次使用C++的用户都会问，为何不能在类定义中初始化成员？现在可以这样做了，其语法类似于下面这样：

```

class Session
{
    int mem1 = 10;           // in-class initialization
    double mem2 {1966.54};  // in-class initialization
    short mem3;

public:
    Session(){}           // #1
    Session(short s) : mem3(s) {}           // #2
    Session(int n, double d, short s) : mem1(n), mem2(d), mem3(s) {} // #3
    ...
};

```

可使用等号或大括号版本的初始化，但不能使用圆括号版本的初始化。其结果与给前两个构造函数提供成员初始化列表，并指定mem1和mem2的值相同：

```
Session() : mem1(10), mem2(1966.54) {}  
Session(short s) : mem1(10), mem2(1966.54), mem3(s) {}
```

通过使用类内初始化，可避免在构造函数中编写重复的代码，从而降低了程序员的工作量、厌倦情绪和出错的机会。

如果构造函数在成员初始化列表中提供了相应的值，这些默认值将被覆盖，因此第三个构造函数覆盖了类内成员初始化。

18.1.8 模板和STL方面的修改

为改善模板和标准模板库的可用性，C++11做了多个改进；有些是库本身，有些与易用性相关。本章前面提到了模板别名和适用于STL的智能指针。

1. 基于范围的**for**循环

对于内置数组以及包含方法begin() 和end() 的类（如std::string）和STL容器，基于范围的for循环（第5章和第16章讨论过）可简化为它们编写循环的工作。这种循环对数组或容器中的每个元素执行指定的操作：

```
double prices[5] = {4.99, 10.99, 6.87, 7.99, 8.49};  
for (double x : prices)  
    std::cout << x << std::endl;
```

其中，x将依次为prices中每个元素的值。x的类型应与数组元素的类型匹配。一种更容易、更安全的方式是，使用auto来声明x，这样编译器将根据prices声明中的信息来推断x的类型：

```
double prices[5] = {4.99, 10.99, 6.87, 7.99, 8.49};  
for (auto x : prices)  
    std::cout << x << std::endl;
```

如果要在循环中修改数组或容器的每个元素，可使用引用类型：

```
std::vector<int> vi(6);
for (auto & x: vi)           // use a reference if loop alters contents
    x = std::rand();
```

2. 新的STL容器

C++11新增了STL容器forward_list、unordered_map、unordered_multimap、unordered_set和unordered_multiset（参见第16章）。容器forward_list是一种单向链表，只能沿一个方向遍历；与双向链接的list容器相比，它更简单，在占用存储空间方面更经济。其他四种容器都是使用哈希表实现的。

C++11还新增了模板array（这在第4和16章讨论过）。要实例化这种模板，可指定元素类型和固定的元素数：

```
std::array<int,360> ar; // array of 360 ints
```

这个模板类没有满足所有的常规模板需求。例如，由于长度固定，您不能使用任何修改容器大小的方法，如put_back()。但array确实有方法begin()和end()，这让您能够对array对象使用众多基于范围的STL算法。

3. 新的STL方法

C++11新增了STL方法cbegin()和cend()。与begin()和end()一样，这些新方法也返回一个迭代器，指向容器的第一个元素和最后一个元素的后面，因此可用于指定包含全部元素的区间。另外，这些新方法将元素视为const。与此类似，crbegin()和crend()是rbegin()和rend()的const版本。

更重要的是，除传统的复制构造函数和常规赋值运算符外，STL容器现在还有移动构造函数和移动赋值运算符。移动语义将在本章后面介绍。

4. valarray升级

模板valarray独立于STL开发的，其最初的设计导致无法将基于范围的STL算法用于valarray对象。C++11添加了两个函数(begin()和end())，它们都接受valarray作为参数，并返回迭代器，这些迭代器分别指

向valarray对象的第一个元素和最后一个元素后面。这让您能够将基于范围的STL算法用于valarray（参见第16章）。

5. 摒弃export

C++98新增了关键字export，旨在提供一种途径，让程序员能够将模板定义放在接口文件和实现文件中，其中前者包含原型和模板声明，而后者包含模板函数和方法的定义。实践证明这不现实，因此C++11终止了这种用法，但仍保留了关键字export，供以后使用。

6. 尖括号

为避免与运算符>>混淆，C++要求在声明嵌套模板时使用空格将尖括号分开：

```
std::vector<std::list<int> > vl; // >> not ok
```

C++11不再这样要求：

```
std::vector<std::list<int>> vl; // >> ok in C++11
```

18.1.9 右值引用

传统的C++引用（现在称为左值引用）使得标识符关联到左值。左值是一个表示数据的表达式（如变量名或解除引用的指针），程序可获取其地址。最初，左值可出现在赋值语句的左边，但修饰符const的出现使得可以声明这样的标识符，即不能给它赋值，但可获取其地址：

```
int n;
int * pt = new int;
const int b = 101; // can't assign to b, but &b is valid
int & rn = n;      // n identifies datum at address &n
int & rt = *pt;    // *pt identifies datum at address pt
const int & rb = b; // b identifies const datum at address &b
```

C++11新增了右值引用（这在第8章讨论过），这是使用&&表示的。右值引用可关联到右值，即可出现在赋值表达式右边，但不能对其应用地址运算符的值。右值包括字面常量（C-风格字符串除外，它表示

地址)、诸如 $x + y$ 等表达式以及返回值的函数(条件是该函数返回的不是引用)：

```
int x = 10;
int y = 23;
int && r1 = 13;
int && r2 = x + y;
double && r3 = std::sqrt(2.0);
```

注意， $r2$ 关联到的是当时计算 $x + y$ 得到的结果。也就是说， $r2$ 关联到的是23，即使以后修改了 x 或 y ，也不会影响到 $r2$ 。

有趣的是，将右值关联到右值引用导致该右值被存储到特定的位置，且可以获取该位置的地址。也就是说，虽然不能将运算符 $\&$ 用于13，但可将其用于 $r1$ 。通过将数据与特定的地址关联，使得可以通过右值引用来访问该数据。

程序清单18.1是一个简短的示例，演示了上述有关右值引用的要点。

程序清单18.1 rvref.cpp

```
// rvref.cpp -- simple uses of rvalue references
#include <iostream>

inline double f(double tf) {return 5.0*(tf-32.0)/9.0;};

int main()
{
    using namespace std;
    double tc = 21.5;
    double && rd1 = 7.07;
    double && rd2 = 1.8 * tc + 32.0;
    double && rd3 = f(rd2);
    cout << " tc value and address: " << tc <<, " << &tc << endl;
    cout << "rd1 value and address: " << rd1 <<, " << &rd1 << endl;
    cout << "rd2 value and address: " << rd2 <<, " << &rd2 << endl;
    cout << "rd3 value and address: " << rd3 <<, " << &rd3 << endl;
    cin.get();
    return 0;
}
```

该程序的输出如下：

```
tc value and address: 21.5, 002FF744
rd1 value and address: 7.07, 002FF728
rd2 value and address: 70.7, 002FF70C
rd3 value and address: 21.5, 002FF6F0
```

引入右值引用的主要目的之一是实现移动语义，这是本章将讨论的下一个主题。

18.2 移动语义和右值引用

现在介绍本书前面未讨论的主题。C++11支持移动语义，这就提出了一些问题：什么是移动语义？C++11如何支持它？为何需要移动语义？下面首先讨论第一个问题。

18.2.1 为何需要移动语义

先来看C++11之前的复制过程。假设有如下代码：

```
vector<string> vstr;
// build up a vector of 20,000 strings, each of 1000 characters
...
vector<string> vstr_copy1(vstr); // make vstr_copy1 a copy of vstr
```

vector和string类都使用动态内存分配，因此它们必须定义使用某种new版本的复制构造函数。为初始化对象vstr_copy1，复制构造函数vector<string>将使用new给20000个string对象分配内存，而每个string对象又将调用string的复制构造函数，该构造函数使用new为1000个字符分配内存。接下来，全部20000000个字符都将从vstr控制的内存中复制到vstr_copy1控制的内存中。这里的工作量很大，但只要妥当就行。

但这确实妥当吗？有时候答案是否定的。例如，假设有一个函数，它返回一个vector<string>对象：

```
vector<string> allcaps(const vector<string> & vs)
{
    vector<string> temp;
    // code that stores an all-uppercase version of vs in temp
    return temp;
}
```

接下来，假设以下面这种方式使用它：

```
vector<string> vstr;
// build up a vector of 20,000 strings, each of 1000 characters
vector<string> vstr_copy1(vstr);           // #1
vector<string> vstr_copy2(allcaps(vstr)); // #2
```

从表面上看，语句#1和#2类似，它们都使用一个现有的对象初始化一个vector<string>对象。如果深入探索这些代码，将发现allcaps()创建了对象temp，该对象管理着20000000个字符；vector和string的复制构造函数创建这20000000个字符的副本，然后程序删除allcaps()返回的临时对象（迟钝的编译器甚至可能将temp复制给一个临时返回对象，删除

`temp`, 再删除临时返回对象）。这里的要点是，做了大量的无用功。考虑到临时对象被删除了，如果编译器将对数据的所有权直接转让给`vstr_copy2`，不是更好吗？也就是说，不将20000000个字符复制到新地方，再删除原来的字符，而将字符留在原来的地方，并将`vstr_copy2`与之相关联。这类似于在计算机中移动文件的情形：实际文件还留在原来的地方，而只修改记录。这种方法被称为移动语义（move semantics）。有点悖论的是，移动语义实际上避免了移动原始数据，而只是修改了记录。

要实现移动语义，需要采取某种方式，让编译器知道什么时候需要复制，什么时候不需要。这就是右值引用发挥作用的地方。可定义两个构造函数。其中一个是常规复制构造函数，它使用`const`左值引用作为参数，这个引用关联到左值实参，如语句#1中的`vstr`；另一个是移动构造函数，它使用右值引用作为参数，该引用关联到右值实参，如语句#2中`allcaps(vstr)`的返回值。复制构造函数可执行深复制，而移动构造函数只调整记录。在将所有权转移给新对象的过程中，移动构造函数可能修改其实参，这意味着右值引用参数不应是`const`。

18.2.2 一个移动示例

下面通过一个示例演示移动语义和右值引用的工作原理。程序清单18.2定义并使用了`Useless`类，这个类动态分配内存，并包含常规复制构造函数和移动构造函数，其中移动构造函数使用了移动语义和右值引用。为演示流程，构造函数和析构函数都比较啰嗦，同时`Useless`类还使用了一个静态变量来跟踪对象数量。另外，省略了一些重要的方法，如赋值运算符。

程序清单18.2 `useless.cpp`

```
// useless.cpp -- an otherwise useless class with move semantics
#include <iostream>
using namespace std;

// interface
class Useless
{
private:
    int n;          // number of elements
    char * pc;      // pointer to data
    static int ct;  // number of objects
```

```

        void ShowObject() const;
public:
    Useless();
    explicit Useless(int k);
    Useless(int k, char ch);
    Useless(const Useless & f); // regular copy constructor
    Useless(Useless && f);      // move constructor
    ~Useless();
    Useless operator+(const Useless & f) const;
// need operator=() in copy and move versions
    void ShowData() const;
};

// implementation
int Useless::ct = 0;

Useless::Useless()
{
    ++ct;
    n = 0;
    pc = nullptr;
    cout << "default constructor called; number of objects: " << ct << endl;
    ShowObject();
}

Useless::Useless(int k) : n(k)
{
    ++ct;
    cout << "int constructor called; number of objects: " << ct << endl;
    pc = new char[n];
    ShowObject();
}

Useless::Useless(int k, char ch) : n(k)
{
    ++ct;
    cout << "int, char constructor called; number of objects: " << ct
        << endl;
    pc = new char[n];
    for (int i = 0; i < n; i++)
        pc[i] = ch;
    ShowObject();
}

Useless::Useless(const Useless & f): n(f.n)
{

```



```

    ++ct;
    cout << "copy const called; number of objects: " << ct << endl;
    pc = new char[n];
    for (int i = 0; i < n; i++)
        pc[i] = f.pc[i];
    ShowObject();
}

Useless::Useless(Useless && f): n(f.n)
{
    ++ct;
    cout << "move constructor called; number of objects: " << ct << endl;
    pc = f.pc;           // steal address
    f.pc = nullptr;     // give old object nothing in return
    f.n = 0;
    ShowObject();
}

Useless::~Useless()
{
    cout << "destructor called; objects left: " << --ct << endl;
    cout << "deleted object:\n";
    ShowObject();
    delete [] pc;
}

Useless Useless::operator+(const Useless & f) const
{
    cout << "Entering operator+()\n";
    Useless temp = Useless(n + f.n);
    for (int i = 0; i < n; i++)
        temp.pc[i] = pc[i];
    for (int i = n; i < temp.n; i++)
        temp.pc[i] = f.pc[i - n];
    cout << "temp object:\n";
    cout << "Leaving operator+()\n";
    return temp;
}

void Useless::ShowObject() const
{
    cout << "Number of elements: " << n;
    cout << " Data address: " << (void *) pc << endl;
}

```



```

void Useless::ShowData() const
{
    if (n == 0)
        cout << "(object empty)";
    else
        for (int i = 0; i < n; i++)
            cout << pc[i];
    cout << endl;
}

// application
int main()
{
{
    Useless one(10, 'x');
    Useless two = one;           // calls copy constructor
    Useless three(20, 'o');
    Useless four (one + three); // calls operator+(), move constructor
    cout << "object one: ";
    one.ShowData();
    cout << "object two: ";
    two.ShowData();
    cout << "object three: ";
    three.ShowData();
    cout << "object four: ";
    four.ShowData();
}
}

```

其中最重要的是复制构造函数和移动构造函数的定义。首先来看复制构造函数（删除了输出语句）：

```
Useless::Useless(const Useless & f): n(f.n)
{
    ++ct;
    pc = new char[n];
    for (int i = 0; i < n; i++)
        pc[i] = f.pc[i];
}
```

它执行深复制，是下面的语句将使用的构造函数：

```
Useless two = one; // calls copy constructor
```

引用f将指向左值对象one。

接下来看移动构造函数，这里也删除了输出语句：

```
Useless::Useless(Useless && f): n(f.n)
{
    ++ct;
    pc = f.pc; // steal address
    f.pc = nullptr; // give old object nothing in return
    f.n = 0;
}
```

它让pc指向现有的数据，以获取这些数据的所有权。此时，pc和f.pc指向相同的数据，调用析构函数时这将带来麻烦，因为程序不能对同一个地址调用delete []两次。为避免这种问题，该构造函数随后将原来的指针设置为空指针，因为对空指针执行delete []没有问题。这种夺取所有权的方式常被称为窃取（pilfering）。上述代码还将原始对象的元素数设置为零，这并非必不可少的，但让这个示例的输出更一致。注意，由于修改了f对象，这要求不能在参数声明中使用const。

在下面的语句中，将使用这个构造函数：

```
Useless four (one + three); // calls move constructor
```

表达式one + three调用Useless::operator+(), 而右值引用f将关联到该方法返回的临时对象。

下面是在Microsoft Visual C++ 2010中编译时，该程序的输出：

```
int, char constructor called; number of objects: 1
Number of elements: 10 Data address: 006F4B68
copy const called; number of objects: 2
Number of elements: 10 Data address: 006F4BB0
int, char constructor called; number of objects: 3
Number of elements: 20 Data address: 006F4BF8
Entering operator+()
int constructor called; number of objects: 4
Number of elements: 30 Data address: 006F4C48
temp object:
Leaving operator+()
move constructor called; number of objects: 5
Number of elements: 30 Data address: 006F4C48
destructor called; objects left: 4
deleted object:
Number of elements: 0 Data address: 00000000
object one: xxxxxxxxxxxx
object two: xxxxxxxxxxxx
object three: ooooooooooooooooooooo
object four: xxxxxxxxxxxxoooooooooooooooo
destructor called; objects left: 3
```

```
deleted object:  
Number of elements: 30 Data address: 006F4C48  
destructor called; objects left: 2  
deleted object:  
Number of elements: 20 Data address: 006F4BF8  
destructor called; objects left: 1  
deleted object:  
Number of elements: 10 Data address: 006F4BB0  
destructor called; objects left: 0  
deleted object:  
Number of elements: 10 Data address: 006F4B68
```

注意到对象two是对象one的副本：它们显示的数据输出相同，但显示的数据地址不同（006F4B68和006F4BB0）。另一方面，在方法Useless::operator+()中创建的对象的数据地址与对象four存储的数据地址相同（都是006F4C48），其中对象four是由移动复制构造函数创建的。另外，注意到创建对象four后，为临时对象调用了析构函数。之所以知道这是临时对象，是因为其元素数和数据地址都是0。

如果使用编译器g++ 4.5.0和标记-std=c++11编译该程序（但将nullptr替换为0），输出将不同，这很有趣：

```
int, char constructor called; number of objects: 1
Number of elements: 10 Data address: 0xa50338
copy const called; number of objects: 2
Number of elements: 10 Data address: 0xa50348
int, char constructor called; number of objects: 3
Number of elements: 20 Data address: 0xa50358
Entering operator+()
int constructor called; number of objects: 4
Number of elements: 30 Data address: 0xa50370
temp object:
Leaving operator+()
object one: xxxxxxxxxxxx
object two: xxxxxxxxxxxx
object three: oooooooooooooooo
object four: xxxxxxxxxxxxooooooooooooooo
destructor called; objects left: 3
deleted object:
Number of elements: 30 Data address: 0xa50370
destructor called; objects left: 2
deleted object:
Number of elements: 20 Data address: 0xa50358
destructor called; objects left: 1
deleted object:
Number of elements: 10 Data address: 0xa50348
destructor called; objects left: 0
deleted object:
Number of elements: 10 Data address: 0xa50338
```

注意到没有调用移动构造函数，且只创建了4个对象。创建对象four时，该编译器没有调用任何构造函数；相反，它推断出对象four是operator+()所做工作的受益人，因此将operator+()创建的对象转到four的名下。一般而言，编译器完全可以进行优化，只要结果与未优化时相同。即使您省略该程序中的移动构造函数，并使用g++进行编译，结果也将相同。

18.2.3 移动构造函数解析

虽然使用右值引用可支持移动语义，但这并不会神奇地发生。要让移动语义发生，需要两个步骤。首先，右值引用让编译器知道何时可使用移动语义：

```
Useless two = one;           // matches Useless::Useless(const Useless &)
Useless four (one + three); // matches Useless::Useless(Useless &&)
```

对象one是左值，与左值引用匹配，而表达式one + three是右值，与右值引用匹配。因此，右值引用让编译器使用移动构造函数来初始化对象four。实现移动语义的第二步是，编写移动构造函数，使其提供所需的行为。

总之，通过提供一个使用左值引用的构造函数和一个使用右值引用的构造函数，将初始化分成了两组。使用左值对象初始化对象时，将使用复制构造函数，而使用右值对象初始化对象时，将使用移动构造函数。程序员可根据需要赋予这些构造函数不同的行为。

这就带来了一个问题：在引入右值引用前，情况是什么样的呢？如果没有移动构造函数，且编译器未能通过优化消除对复制构造函数的需求，结果将如何呢？在C++98中，下面的语句将调用复制构造函数：

```
Useless four (one + three);
```

但左值引用不能指向右值。结果将如何呢？第8章介绍过，如果实参为右值，const引用形参将指向一个临时变量：

```
int twice(const & rx) {return 2 * rx; }

...
int main()
{
    int m = 6;
    // below, rx refers to m
    int n = twice(m);
    // below, rx refers to a temporary variable initialized to 21
    int k = twice(21);
    ...
}
```

就Useless而言，形参f将被初始化一个临时对象，而该临时对象被初始化为operator+()返回的值。下面是使用老式编译器进行编译时，程序清单18.2所示程序（删除了移动构造函数）的部分输出：

```
...
Entering operator+()
int constructor called; number of objects: 4
Number of elements: 30 Data address: 01C337C4
temp object:
Leaving operator+()
copy const called; number of objects: 5
Number of elements: 30 Data address: 01C337E8
destructor called; objects left: 4
deleted object:
Number of elements: 30 Data address: 01C337C4
copy const called; number of objects: 5
Number of elements: 30 Data address: 01C337C4
destructor called; objects left: 4
deleted object:
Number of elements: 30 Data address: 01C337E8
...

```

首先，在方法Useless::operator+() 内，调用构造函数创建了temp，并在01C337C4处给它分配了存储30个元素的空间。然后，调用复制构造函数创建了一个临时复制信息（其地址为01C337E8），f指向该副本。接下来，删除了地址为01C337C4的对象temp。然后，新建了对象four，它使用了01C337C4处刚释放的内存。接下来，删除了01C337E8处的临时参数对象。这表明，总共创建了三个对象，但其中的两个被删除。这些就是移动语义旨在消除的额外工作。

正如g++示例表明的，机智的编译器可能自动消除额外的复制工作，但通过使用右值引用，程序员可指出何时该使用移动语义。

18.2.4 赋值

适用于构造函数的移动语义考虑也适用于赋值运算符。例如，下面演示了如何给Useless类编写复制赋值运算符和移动赋值运算符：

```
Useless & Useless::operator=(const Useless & f) // copy assignment
{
    if (this == &f)
        return *this;
    delete [] pc;
    n = f.n;
    pc = new char[n];
    for (int i = 0; i < n; i++)
        pc[i] = f.pc[i];
    return *this;
}

Useless & Useless::operator=(Useless && f) // move assignment
{
    if (this == &f)
        return *this;
    delete [] pc;
    n = f.n;
    pc = f.pc;
    f.n = 0;
    f.pc = nullptr;
    return *this;
}
```

上述复制赋值运算符采用了第12章介绍的常规模式，而移动赋值运算符删除目标对象中的原始数据，并将源对象的所有权转让给目标。不能让多个指针指向相同的数据，这很重要，因此上述代码将源对象中的指针设置为空指针。

与移动构造函数一样，移动赋值运算符的参数也不能是const引用，因为这个方法修改了源对象。

18.2.5 强制移动

移动构造函数和移动赋值运算符使用右值。如果要让它们使用左值，该如何办呢？例如，程序可能分析一个包含候选对象的数组，选择其中一个对象供以后使用，并丢弃数组。如果可以使用移动构造函数或移动赋值运算符来保留选定的对象，那该多好啊。然而，假设您试图像下面这样做：

```
Useless choices[10];
Useless best;
int pick;
... // select one object, set pick to index
best = choices[pick];
```

由于choices[pick]是左值，因此上述赋值语句将使用复制赋值运算符，而不是移动赋值运算符。但如果能让choices[pick]看起来像右值，便将使用移动赋值运算符。为此，可使用运算符static_cast<>将对象的类型强制转换为Useless &&，但C++11提供了一种更简单的方式—使用头文件utility中声明的函数std::move()。程序清单18.3演示了这种技术，它在Useless类中添加了啰嗦的赋值运算符，并让以前啰嗦的构造函数和析构函数保持沉默。

程序清单18.3 stdmove.cpp

```
// stdmove.cpp -- using std::move()
#include <iostream>
#include <utility>

// interface
class Useless
{
private:
    int n;           // number of elements
    char * pc;       // pointer to data
    static int ct;   // number of objects
    void ShowObject() const;
public:
    Useless();
    explicit Useless(int k);
    Useless(int k, char ch);
    Useless(const Useless & f); // regular copy constructor
    Useless(Useless && f);     // move constructor
    ~Useless();
    Useless operator+(const Useless & f) const;
    Useless & operator=(const Useless & f); // copy assignment
    Useless & operator=(Useless && f);      // move assignment
    void ShowData() const;
};

// implementation
int Useless::ct = 0;

Useless::Useless()
{
    ++ct;
    n = 0;
    pc = nullptr;
}

Useless::Useless(int k) : n(k)
{
    ++ct;
    pc = new char[n];
}
```



```

Useless::Useless(int k, char ch) : n(k)
{
    ++ct;
    pc = new char[n];
    for (int i = 0; i < n; i++)
        pc[i] = ch;
}

Useless::Useless(const Useless & f): n(f.n)
{
    ++ct;
    pc = new char[n];
    for (int i = 0; i < n; i++)
        pc[i] = f.pc[i];
}

Useless::Useless(Useless && f): n(f.n)
{
    ++ct;
    pc = f.pc;           // steal address
    f.pc = nullptr;     // give old object nothing in return
    f.n = 0;
}

Useless::~Useless()
{
    delete [] pc;
}

Useless & Useless::operator=(const Useless & f) // copy assignment
{
    std::cout << "copy assignment operator called:\n";
    if (this == &f)
        return *this;
    delete [] pc;
    n = f.n;
    pc = new char[n];
    for (int i = 0; i < n; i++)
        pc[i] = f.pc[i];
    return *this;
}

Useless & Useless::operator=(Useless && f)      // move assignment
{
    std::cout << "move assignment operator called:\n";
    if (this == &f)

```



```

        return *this;
    delete [] pc;
    n = f.n;
    pc = f.pc;
    f.n = 0;
    f.pc = nullptr;
    return *this;
}

Useless Useless::operator+(const Useless & f) const
{
    Useless temp = Useless(n + f.n);
    for (int i = 0; i < n; i++)
        temp.pc[i] = pc[i];
    for (int i = n; i < temp.n; i++)
        temp.pc[i] = f.pc[i - n];
    return temp;
}

void Useless::ShowObject() const
{
    std::cout << "Number of elements: " << n;
    std::cout << " Data address: " << (void *) pc << std::endl;
}

void Useless::ShowData() const
{
    if (n == 0)
        std::cout << "(object empty)";
    else
        for (int i = 0; i < n; i++)
            std::cout << pc[i];
    std::cout << std::endl;
}

// application
int main()
{
    using std::cout;
    {
        Useless one(10, 'x');
        Useless two = one +one; // calls move constructor
        cout << "object one: ";
        one.ShowData();
        cout << "object two: ";
        two.ShowData();
    }
}

```

```
Useless three, four;
cout << "three = one\n";
three = one;           // automatic copy assignment
cout << "now object three = ";
three.ShowData();
cout << "and object one = ";
one.ShowData();
cout << "four = one + two\n";
four = one + two;      // automatic move assignment
cout << "now object four = ";
four.ShowData();
cout << "four = move(one)\n";
four = std::move(one); // forced move assignment
cout << "now object four = ";
four.ShowData();
cout << "and object one = ";
one.ShowData();
}
}
```

该程序的输出如下：

```
object one: xxxxxxxxxxxx
object two: xxxxxxxxxxxxxxxxxxxxxxxxx
three = one
copy assignment operator called:
now object three = xxxxxxxxxxxx
and object one = xxxxxxxxxxxx
four = one + two
move assignment operator called:
now object four = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
four = move(one)
move assignment operator called:
now object four = xxxxxxxxxxxx
and object one = (object empty)
```

正如您看到的，将one赋给three调用了复制赋值运算符，但将move(one)赋给four调用的是移动赋值运算符。

需要知道的是，函数std::move()并非一定会导致移动操作。例如，假设Chunk是一个包含私有数据的类，而您编写了如下代码：

```
Chunk one;
...
Chunk two;
two = std::move(one); // move semantics?
```

表达式std::move(one)是右值，因此上述赋值语句将调用Chunk的移动赋值运算符—如果定义了这样的运算符。但如果Chunk没有定义移动赋值运算符，编译器将使用复制赋值运算符。如果没有定义复制赋值运算符，将根本不允许上述赋值。

对大多数程序员来说，右值引用带来的主要好处并非是让他们能够

编写使用右值引用的代码，而是能够使用利用右值引用实现移动语义的库代码。例如，STL类现在都有复制构造函数、移动构造函数、复制赋值运算符和移动赋值运算符。

18.3 新的类功能

除本章前面提到的显式转换运算符和类内成员初始化外，C++11还新增了其他几个类功能。

18.3.1 特殊的成员函数

在原有4个特殊成员函数（默认构造函数、复制构造函数、复制赋值运算符和析构函数）的基础上，C++11新增了两个：移动构造函数和移动赋值运算符。这些成员函数是编译器在各种情况下自动提供的。

前面说过，在没有提供任何参数的情况下，将调用默认构造函数。如果您没有给类定义任何构造函数，编译器将提供一个默认构造函数。这种版本的默认构造函数被称为默认的默认构造函数。对于使用内置类型的成员，默认的默认构造函数不对其进行初始化；对于属于类对象的成员，则调用其默认构造函数。

另外，如果您没有提供复制构造函数，而代码又需要使用它，编译器将提供一个默认的复制构造函数；如果您没有提供移动构造函数，而代码又需要使用它，编译器将提供一个默认的移动构造函数。假定类名为Someclass，这两个默认的构造函数的原型如下：

```
Someclass::Someclass(const Someclass &); // defaulted copy constructor  
Someclass::Someclass(Someclass &&); // defaulted move constructor
```

在类似的情况下，编译器将提供默认的复制运算符和默认的移动运算符，它们的原型如下：

```
Someclass & Someclass::operator(const Someclass &); // defaulted copy assignment  
Someclass & Someclass::operator(Someclass &&); // defaulted move assignment
```

最后，如果您没有提供析构函数，编译器将提供一个。

对于前面描述的情况，有一些例外。如果您提供了析构函数、复制

构造函数或复制赋值运算符，编译器将不会自动提供移动构造函数和移动赋值运算符；如果您提供了移动构造函数或移动赋值运算符，编译器将不会自动提供复制构造函数和复制赋值运算符。

另外，默认的移动构造函数和移动赋值运算符的工作方式与复制版本类似：执行逐成员初始化并复制内置类型。如果成员是类对象，将使用相应类的构造函数和赋值运算符，就像参数为右值一样。如果定义了移动构造函数和移动赋值运算符，这将调用它们；否则将调用复制构造函数和复制赋值运算符。

18.3.2 默认的方法和禁用的方法

C++11让您能够更好地控制要使用的方法。假定您要使用某个默认的函数，而这个函数由于某种原因不会自动创建。例如，您提供了移动构造函数，因此编译器不会自动创建默认的构造函数、复制构造函数和复制赋值构造函数。在这些情况下，您可使用关键字default显式地声明这些方法的默认版本：

```
class Someclass
{
public:
    Someclass(Someclass &&);
    Someclass() = default;           // use compiler-generated default constructor
    Someclass(const Someclass &) = default;
    Someclass & operator=(const Someclass &) = default;
    ...
};
```

编译器将创建在您没有提供移动构造函数的情况下将自动提供的构造函数。

另一方面，关键字delete可用于禁止编译器使用特定方法。例如，要禁止复制对象，可禁用复制构造函数和复制赋值运算符：

```

class Someclass
{
public:
    Someclass() = default;      // use compiler-generated default constructor
// disable copy constructor and copy assignment operator:
    Someclass(const Someclass &) = delete;
    Someclass & operator=(const Someclass &) = delete;
// use compiler-generated move constructor and move assignment operator:
    Someclass(Someclass &&) = default;
    Someclass & operator=(Someclass &&) = default;
    Someclass & operator+(const Someclass &) const;
...
};

```

第12章说过，要禁止复制，可将复制构造函数和赋值运算符放在类定义的private部分，但使用delete也能达到这个目的，且更不容易犯错、更容易理解。

如果在启用移动方法的同时禁用复制方法，结果将如何呢？前面说过，移动操作使用的右值引用只能关联到右值表达式，这意味着：

```

Someclass one;
Someclass two;
Someclass three(one);      // not allowed, one an lvalue
Someclass four(one + two); // allowed, expression is an rvalue

```

关键字default只能用于6个特殊成员函数，但delete可用于任何成员函数。delete的一种可能用法是禁止特定的转换。例如，假设Someclass类有一个接受double参数的方法：

```

class Someclass
{
public:
...
    void redo(double);
...
};

```

再假设有如下代码：

```
Someclass sc;  
sc.redo(5);
```

int值5将被提升为5.0，进而执行方法redo()。

现在假设将Someclass类的定义改成了下面这样：

```
class Someclass  
{  
public:  
    ...  
    void redo(double);  
    void redo(int) = delete;  
    ...  
};
```

在这种情况下，方法调用sc.redo(5)与原型redo(int)匹配。编译器检测到这一点以及redo(int)被禁用后，将这种调用视为编译错误。这说明了禁用函数的重要一点：它们只用于查找匹配函数，使用它们将导致编译错误。

18.3.3 委托构造函数

如果给类提供了多个构造函数，您可能重复编写相同的代码。也就是说，有些构造函数可能需要包含其他构造函数中已有的代码。为让编码工作更简单、更可靠，C++11允许您在一个构造函数的定义中使用另一个构造函数。这被称为委托，因为构造函数暂时将创建对象的工作委托给另一个构造函数。委托使用成员初始化列表语法的变种：

```

class Notes {
    int k;
    double x;
    std::string st;
public:
    Notes();
    Notes(int);
    Notes(int, double);
    Notes(int, double, std::string);
};

Notes::Notes(int kk, double xx, std::string stt) : k(kk),
    x(xx), st(stt) /*do stuff*/
Notes::Notes() : Notes(0, 0.01, "Oh") /* do other stuff*/
Notes::Notes(int kk) : Notes(kk, 0.01, "Ah") /* do yet other stuff*/
Notes::Notes( int kk, double xx ) : Notes(kk, xx, "Uh") /* ditto*/

```

例如，上述默认构造函数使用第一个构造函数初始化数据成员并执行其函数体，然后再执行自己的函数体。

18.3.4 继承构造函数

为进一步简化编码工作，C++11提供了一种让派生类能够继承基类构造函数的机制。C++98提供了一种让名称空间中函数可用的语法：

```

namespace Box
{
    int fn(int) { ... }
    int fn(double) { ... }
    int fn(const char *) { ... }
}

using Box::fn;

```

这让函数fn的所有重载版本都可用。也可使用这种方法让基类的所有非特殊成员函数对派生类可用。例如，请看下面的代码：

```
class C1
{
...
public:
...
    int fn(int j) { ... }
    double fn(double w) { ... }
    void fn(const char * s) { ... }
};

class C2 : public C1
{
...
public:
...
    using C1::fn;
    double fn(double) { ... };
};

...
C2 c2;
int k = c2.fn(3);           // uses C1::fn(int)
double z = c2.fn(2.4);     // uses C2::fn(double)
```

C2中的using声明让C2对象可使用C1的三个fn()方法，但将选择C2而不是C1定义的方法fn(double)。

C++11将这种方法用于构造函数。这让派生类继承基类的所有构造函数（默认构造函数、复制构造函数和移动构造函数除外），但不会使用与派生类构造函数的特征标匹配的构造函数：

```

class BS
{
    int q;
    double w;
public:
    BS() : q(0), w(0) {}
    BS(int k) : q(k), w(100) {}
    BS(double x) : q(-1), w(x) {}
    BS(int k, double x) : q(k), w(x) {}
    void Show() const {std::cout << q << ", " << w << '\n';}
};

class DR : public BS
{
    short j;
public:
    using BS::BS;
    DR() : j(-100) {} // DR needs its own default constructor
    DR(double x) : BS(2*x), j(int(x)) {}
    DR(int i) : j(-2), BS(i, 0.5* i) {}
    void Show() const {std::cout << j << ", "; BS::Show();}
};

int main()
{
    DR o1;           // use DR()
    DR o2(18.81);   // use DR(double) instead of BS(double)
    DR o3(10, 1.8); // use BS(int, double)
    ...
}

```

由于没有构造函数DR(int, double)，因此创建DR对象o3时，将使用继承而来的BS(int, double)。请注意，继承的基类构造函数只初始化基类成员；如果还要初始化派生类成员，则应使用成员列表初始化语法：

```
DR(int i, int k, double x) : j(i), BS(k, x) {}
```

18.3.5 管理虚方法：override和final

虚方法对实现多态类层次结构很重要，让基类引用或指针能够根据指向的对象类型调用相应的方法，但虚方法也带来了一些编程陷阱。例如，假设基类声明了一个虚方法，而您决定在派生类中提供不同的版本，这将覆盖旧版本。但正如第13章讨论的，如果特征标不匹配，将隐藏而不是覆盖旧版本：

```
class Action
{
    int a;
public:
    Action(int i = 0) : a(i) {}
    int val() const {return a;}
    virtual void f(char ch) const { std::cout << val() << ch << "\n"; }
};

class Bingo : public Action
{
public:
    Bingo(int i = 0) : Action(i) {}
    virtual void f(char * ch) const { std::cout << val() << ch << "!\\n"; }
};
```

由于类Bingo定义的是f(char * ch)而不是f(char ch)，将对Bingo对象隐藏f(char ch)，这导致程序不能使用类似于下面的代码：

```
Bingo b(10);
b.f('@'); // works for Action object, fails for Bingo object
```

在C++11中，可使用虚说明符override指出您要覆盖一个虚函数：将其放在参数列表后面。如果声明与基类方法不匹配，编译器将视为错误。因此，下面的Bingo::f()版本将生成一条编译错误消息：

```
virtual void f(char * ch) const override { std::cout << val()
                                            << ch << "!\\n"; }
```

例如，在Microsoft Visual C++ 2010中，出现的错误消息如下：

```
method with override specifier 'override' did not override any
base class methods
```

说明符final解决了另一个问题。您可能想禁止派生类覆盖特定的虚方法，为此可在参数列表后面加上final。例如，下面的代码禁止Action的派生类重新定义函数f()：

```
virtual void f(char ch) const final { std::cout << val() << ch << "\n"; }
```

说明符override和final并非关键字，而是具有特殊含义的标识符。这意味着编译器根据上下文确定它们是否有特殊含义；在其他上下文中，可将它们用作常规标识符，如变量名或枚举。

18.4 Lambda函数

见到术语lambda函数（也叫lambda表达式，常简称为lambda）时，您可能怀疑C++11添加这项新功能旨在帮助编程新手。看到下面的lambda函数示例后，您可能坚定了自己的怀疑：

```
[&count] (int x) {count += (x % 13 == 0);}
```

但lambda函数并不像看起来那么晦涩难懂，它们提供了一种有用的服务，对使用函数谓词的STL算法来说尤其如此。

18.4.1 比较函数指针、函数符和Lambda函数

来看一个示例，它使用三种方法给STL算法传递信息：函数指针、函数符和lambda。出于方便的考虑，将这三种形式通称为函数对象，以免不断地重复“函数指针、函数符或lambda”。假设您要生成一个随机整数列表，并判断其中多少个整数可被3整除，多少个整数可被13整除。

生成这样的列表很简单。一种方案是，使用vector<int>存储数字，并使用STL算法generate() 在其中填充随机数：

```
#include <vector>
#include <algorithm>
#include <cmath>
...
std::vector<int> numbers(1000);
std::generate(vector.begin(), vector.end(), std::rand);
```

函数generate()接受一个区间（由前两个参数指定），并将每个元素设置为第三个参数返回的值，而第三个参数是一个不接受任何参数的函数对象。在上述示例中，该函数对象是一个指向标准函数rand()的指针。

通过使用算法count_if()，很容易计算出有多少个元素可被3整除。与函数generate()一样，前两个参数应指定区间，而第三个参数应是一个返回true或false的函数对象。函数count_if()计算这样的元素数，即它使得指定的函数对象返回true。为判断元素能否被3整除，可使用下面的函数定义：

```
bool f3(int x) {return x % 3 == 0;}
```

同样，为判断元素能否被13整除，可使用下面的函数定义：

```
bool f13(int x) {return x % 13 == 0;}
```

定义上述函数后，便可计算复合条件的元素数了，如下所示：

```
int count3 = std::count_if(numbers.begin(), numbers.end(), f3);
cout << "Count of numbers divisible by 3: " << count3 << '\n';
int count13 = std::count_if(numbers.begin(), numbers.end(), f13);
cout << "Count of numbers divisible by 13: " << count13 << "\n\n";
```

下面复习一下如何使用函数符来完成这个任务。第16章介绍过，函数符是一个类对象，并非只能像函数名那样使用它，这要归功于类方法operator()()。就这个示例而言，函数符的优点之一是，可使用同一个函数符来完成这两项计数任务。下面是一种可能的定义：

```

class f_mod
{
private:
    int dv;
public:
    f_mod(int d = 1) : dv(d) {}
    bool operator()(int x) {return x % dv == 0;}
};

```

这为何可行呢？因为可使用构造函数创建存储特定整数值的f_mod对象：

```
f_mod obj(3); // f_mod.dv set to 3
```

而这个对象可使用方法operator()来返回一个bool值：

```
bool is_div_by_3 = obj(7); // same as obj.operator()(7)
```

构造函数本身可用作诸如count_if()等函数的参数：

```
count3 = std::count_if(numbers.begin(), numbers.end(), f_mod(3));
```

参数f_mod(3)创建一个对象，它存储了值3；而count_if()使用该对象来调用operator()()，并将参数x设置为numbers的一个元素。要计算有多少个数字可被13（而不是3）整除，只需将第三个参数设置为f_mod(3)。

最后，来看看使用lambda的情况。名称lambda来自lambda calculus（ λ 演算）——一种定义和应用函数的数学系统。这个系统让您能够使用匿名函数——即无需给函数命名。在C++11中，对于接受函数指针或函数符的函数，可使用匿名函数定义（lambda）作为其参数。与前述函数f3()对应的lambda如下：

```
[](int x) {return x % 3 == 0;}
```

这与f3()的函数定义很像：

```
bool f3(int x) {return x % 3 == 0;}
```

差别有两个：使用[]替代了函数名（这就是匿名的由来）；没有声明返回类型。返回类型相当于使用decltype根据返回值推断得到的，这里为bool。如果lambda不包含返回语句，推断出的返回类型将为void。就这个示例而言，您将以如下方式使用该lambda：

```
count3 = std::count_if(numbers.begin(), numbers.end(),
    [] (int x){return x % 3 == 0;});
```

也就是说，使用整个lambad表达式替换函数指针或函数符构造函数。

仅当lambad表达式完全由一条返回语句组成时，自动类型推断才管用；否则，需要使用新增的返回类型后置语法：

```
[] (double x)->double{int y = x; return x - y;} // return type is double
```

程序清单18.4演示了前面讨论的各个要点。

程序清单18.4 lambda0.cpp

```
// lambda0.cpp -- using lambda expressions
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <ctime>
const long Size1 = 39L;
const long Size2 = 100*Size1;
const long Size3 = 100*Size2;
```

```

bool f3(int x) {return x % 3 == 0;}
bool f13(int x) {return x % 13 == 0;}

int main()
{
    using std::cout;
    std::vector<int> numbers(Size1);

    std::srand(std::time(0));
    std::generate(numbers.begin(), numbers.end(), std::rand);

// using function pointers
    cout << "Sample size = " << Size1 << '\n';

    int count3 = std::count_if(numbers.begin(), numbers.end(), f3);
    cout << "Count of numbers divisible by 3: " << count3 << '\n';
    int count13 = std::count_if(numbers.begin(), numbers.end(), f13);
    cout << "Count of numbers divisible by 13: " << count13 << "\n\n";

// increase number of numbers
    numbers.resize(Size2);
    std::generate(numbers.begin(), numbers.end(), std::rand);
    cout << "Sample size = " << Size2 << '\n';
// using a functor
    class f_mod
    {
    private:
        int dv;
    public:
        f_mod(int d = 1) : dv(d) {}
        bool operator()(int x) {return x % dv == 0;}
    };

    count3 = std::count_if(numbers.begin(), numbers.end(), f_mod(3));
    cout << "Count of numbers divisible by 3: " << count3 << '\n';
    count13 = std::count_if(numbers.begin(), numbers.end(), f_mod(13));
    cout << "Count of numbers divisible by 13: " << count13 << "\n\n";

// increase number of numbers again
    numbers.resize(Size3);
    std::generate(numbers.begin(), numbers.end(), std::rand);
    cout << "Sample size = " << Size3 << '\n';
// using lambdas
    count3 = std::count_if(numbers.begin(), numbers.end(),
        [] (int x){return x % 3 == 0;});
    cout << "Count of numbers divisible by 3: " << count3 << '\n';

```

```
count13 = std::count_if(numbers.begin(), numbers.end(),
    [] (int x){return x % 13 == 0;});
cout << "Count of numbers divisible by 13: " << count13 << '\n';

return 0;
}
```

下面是该程序的输出示例：

```
Sample size = 39
Count of numbers divisible by 3: 15
Count of numbers divisible by 13: 6

Sample size = 3900
Count of numbers divisible by 3: 1305
Count of numbers divisible by 13: 302

Sample size = 390000
Count of numbers divisible by 3: 130241
Count of numbers divisible by 13: 29860
```

输出表明，样本很小时，得到的统计数据并不可靠。

18.4.2 为何使用lambda

您可能会问，除那些表达式狂热爱好者，谁会使用lambda呢？下面从4个方面探讨这个问题：距离、简洁、效率和功能。

很多程序员认为，让定义位于使用的地方附近很有用。这样，就无需翻阅多页的源代码，以了解函数调用count_if()的第三个参数了。另外，如果需要修改代码，涉及的内容都将在附近；而剪切并粘贴代码以便在其他地方使用时，涉及的内容也在一起。从这种角度看，lambda是

理想的选择，因为其定义和使用是在同一个地方进行的；而函数是最糟糕的选择，因为不能在函数内部定义其他函数，因此函数的定义可能离使用它的地方很远。函数符是不错的选择，因为可在函数内部定义类（包含函数符类），因此定义离使用地点可以很近。

从简洁的角度看，函数符代码比函数和lambda代码更繁琐。函数和lambda的简洁程度相当，一个显而易见的例外是，需要使用同一个lambda两次：

```
count1 = std::count_if(n1.begin(), n1.end(),
    [] (int x){return x % 3 == 0;});
count2 = std::count_if(n2.begin(), n2.end(),
    [] (int x){return x % 3 == 0;});
```

但并非必须编写lambda两次，而可给lambda指定一个名称，并使用该名称两次：

```
auto mod3 = [] (int x){return x % 3 == 0;} // mod3 a name for the lambda
count1 = std::count_if(n1.begin(), n1.end(), mod3);
count2 = std::count_if(n2.begin(), n2.end(), mod3);
```

您甚至可以像使用常规函数那样使用有名称的lambda：

```
bool result = mod3(z); // result is true if z % 3 == 0
```

然而，不同于常规函数，可在函数内部定义有名称的lambda。mod3的实际类型随实现而异，它取决于编译器使用什么类型来跟踪lambda。

这三种方法的相对效率取决于编译器内联那些东西。函数指针方法阻止了内联，因为编译器传统上不会内联其地址被获取的函数，因为函数地址的概念意味着非内联函数。而函数符和lambda通常不会阻止内联。

最后，lambda有一些额外的功能。具体地说，lambda可访问作用域内的任何动态变量；要捕获要使用的变量，可将其名称放在中括号内。如果只指定了变量名，如[z]，将按值访问变量；如果在名称前加上&，如[&count]，将按引用访问变量。 [&]让您能够按引用访问所有动态变

量，而[=]让您能够按值访问所有动态变量。还可混合使用这两种方式，例如，[ted, &ed]让您能够按值访问ted以及按引用访问ed，[&, ted]让您能够按值访问ted以及按引用访问其他所有动态变量，[=, &ed]让您能够按引用访问ed以及按值访问其他所有动态变量。在程序清单18.4中，可将下述代码：

```
int count13;  
...  
count13 = std::count_if(numbers.begin(), numbers.end(),  
    [](int x){return x % 13 == 0;});
```

替换为如下代码：

```
int count13 = 0;  
std::for_each(numbers.begin(), numbers.end(),  
    [&count13](int x){count13 += x % 13 == 0;});
```

[&count13]让lambda能够在其代码中使用count13。由于count13是按引用捕获的，因此在lambda对count13所做的任何修改都将影响原始count13。如果x能被13整除，则表达式 $x \% 13 == 0$ 将为true，添加到count13中时，true将被转换为1。同样，false将被转换为0。因此，for_each()将lambda应用于numbers的每个元素后，count13将为能被13整除的元素数。

通过利用这种技术，可使用一个lambda表达式计算可被3整除的元素数和可被13整除的元素数：

```
int count3 = 0;  
int count13 = 0;  
std::for_each(numbers.begin(), numbers.end(),  
    [&](int x){count3 += x % 3 == 0; count13 += x % 13 == 0;});
```

在这里，[&]让您能够在lambda表达式中使用所有的自动变量，包括count3和count13。

程序清单18.5演示了如何使用这些技术。

程序清单 18.5 lambda1.cpp

```

// lambda1.cpp -- use captured variables
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <ctime>
const long Size = 390000L;

int main()
{
    using std::cout;
    std::vector<int> numbers(Size);

    std::srand(std::time(0));
    std::generate(numbers.begin(), numbers.end(), std::rand);
    cout << "Sample size = " << Size << '\n';
// using lambdas
    int count3 = std::count_if(numbers.begin(), numbers.end(),
        [] (int x){return x % 3 == 0;});
    cout << "Count of numbers divisible by 3: " << count3 << '\n';
    int count13 = 0;
    std::for_each(numbers.begin(), numbers.end(),
        [&count13] (int x){count13 += x % 13 == 0;});
    cout << "Count of numbers divisible by 13: " << count13 << '\n';
// using a single lambda
    count3 = count13 = 0;
    std::for_each(numbers.begin(), numbers.end(),
        [&] (int x){count3 += x % 3 == 0; count13 += x % 13 == 0;});
    cout << "Count of numbers divisible by 3: " << count3 << '\n';
    cout << "Count of numbers divisible by 13: " << count13 << '\n';
    return 0;
}

```

下面是该程序的示例输出：

```
Sample size = 390000
Count of numbers divisible by 3: 130274
Count of numbers divisible by 13: 30009
Count of numbers divisible by 3: 130274
Count of numbers divisible by 13: 30009
```

输出表明，该程序使用的两种方法（两个独立的lambda和单个lambda）的结果相同。

在C++中引入lambda的主要目的是，让您能够将类似于函数的表达式用作接受函数指针或函数符的函数的参数。因此，典型的lambda是测试表达式或比较表达式，可编写为一条返回语句。这使得lambda简洁而易于理解，且可自动推断返回类型。然而，有创意的C++程序员可能开发出其他用法。

18.5 包装器

C++提供了多个包装器（wrapper，也叫适配器[adapter]）。这些对象用于给其他编程接口提供更一致或更合适的接口。例如，第16章讨论了bind1st和bind2nd，它们让接受两个参数的函数能够与这样的STL算法匹配，即它要求将接受一个参数的函数作为参数。C++11提供了其他的包装器，包括模板bind、mem_fn和reference_wrapper以及包装器function。其中模板bind可替代bind1st和bind2nd，但更灵活；模板mem_fn让您能够将成员函数作为常规函数进行传递；模板reference_wrapper让您能够创建行为像引用但可被复制的对象；而包装器function让您能够以统一的方式处理多种类似于函数的形式。

下面更详细地介绍包装器function及其解决的问题。

18.5.1 包装器function及模板的低效性

请看下面的代码行：

```
answer = ef(q);
```

`ef`是什么呢？它可以是函数名、函数指针、函数对象或有名称的`lambda`表达式。所有这些都是可调用的类型（*callable type*）。鉴于可调用的类型如此丰富，这可能导致模板的效率极低。为明白这一点，来看一个简单的案例。

首先，在头文件中定义一些模板，如程序清单18.6所示。

程序清单18.6 somedefs.h

```
// somedefs.h
#include <iostream>

template <typename T, typename F>
T use_f(T v, F f)
{
    static int count = 0;
    count++;
    std::cout << " use_f count = " << count
           << ", &count = " << &count << std::endl;
    return f(v);
}

class Fp
{
private:
    double z_;
public:
    Fp(double z = 1.0) : z_(z) {}
    double operator()(double p) { return z_*p; }
};

class Fq
{
private:
    double z_;
public:
    Fq(double z = 1.0) : z_(z) {}
    double operator()(double q) { return z_+ q; }
};
```

模板use_f使用参数f表示调用类型：

```
return f(v);
```

接下来，程序清单18.7所示的程序调用模板函数use_f()6次。

程序清单18.7 callable.cpp

```
// callable.cpp -- callable types and templates
#include "somedefs.h"
#include <iostream>

double dub(double x) {return 2.0*x;}
double square(double x) {return x*x;}

int main()
{
    using std::cout;
    using std::endl;

    double y = 1.21;
    cout << "Function pointer dub:\n";
    cout << " " << use_f(y, dub) << endl;
    cout << "Function pointer square:\n";
    cout << " " << use_f(y, square) << endl;
    cout << "Function object Fp:\n";
    cout << " " << use_f(y, Fp(5.0)) << endl;
    cout << "Function object Fq:\n";
    cout << " " << use_f(y, Fq(5.0)) << endl;
    cout << "Lambda expression 1:\n";
    cout << " " << use_f(y, [] (double u) {return u*u;}) << endl;
    cout << "Lambda expression 2:\n";
    cout << " " << use_f(y, [] (double u) {return u+u/2.0;}) << endl;
    return 0;
}
```

在每次调用中，模板参数T都被设置为类型double。模板参数F呢？每次调用时，F都接受一个double值并返回一个double值，因此在6次use_of()调用中，好像F的类型都相同，因此只会实例化模板一次。但正如下面的输出表明的，这种想法太天真了：

```
Function pointer dub:  
use_f count = 1, &count = 0x402028  
2.42  
Function pointer square:  
use_f count = 2, &count = 0x402028  
1.1  
Function object Fp:  
use_f count = 1, &count = 0x402020  
6.05  
Function object Fq:  
use_f count = 1, &count = 0x402024  
6.21  
Lambda expression 1:  
use_f count = 1, &count = 0x405020  
1.4641  
Lambda expression 2:  
use_f count = 1, &count = 0x40501c  
1.815
```

模板函数use_f()有一个静态成员count，可根据它的地址确定模板实例化了多少次。有5个不同的地址，这表明模板use_f()有5个不同的实例化。

为了解其中的原因，请考虑编译器如何判断模板参数F的类型。首先，来看下面的调用：

```
use_f(y, dub);
```

其中的dub是一个函数的名称，该函数接受一个double参数并返回一个double值。函数名是指针，因此参数F的类型为double(*) (double)：一个指向这样的函数的指针，即它接受一个double参数并返回一个double值。

下一个调用如下：

```
use_f(y, square);
```

第二个参数的类型也是double(*) (double)，因此该调用使用的use_f()实例化与第一个调用相同。

在接下来的两个use_f()调用中，第二个参数为对象，F的类型分别为Fp和Fq，因为将为这些F值实例化use_f()模板两次。最后，最后两个调用将F的类型设置为编译器为lambda表达式使用的类型。

18.5.2 修复问题

包装器function让您能够重写上述程序，使其只使用use_f()的一个实例而不是5个。注意到程序清单18.7中的函数指针、函数对象和lambda表达式有一个相同的地方，它们都接受一个double参数并返回一个double值。可以说它们的调用特征标（call signature）相同。调用特征标是有返回类型以及用括号括起并用头号分隔的参数类型列表定义的，因此，这六个实例的调用特征标都是double (double)。

模板function是在头文件functional中声明的，它从调用特征标的角
度定义了一个对象，可用于包装调用特征标相同的函数指针、函数对象或lambda表达式。例如，下面的声明创建一个名为fdci的function对象，它接受一个char参数和一个int参数，并返回一个double值：

```
std::function<double(char, int)> fdci;
```

然后，可以将接受一个char参数和一个int参数，并返回一个double值的任何函数指针、函数对象或lambda表达式赋给它。

在程序清单18.7中，所有可调用参数的调用特征标都相同：double (double)。要修复程序清单18.7以减少实例化次数，可使用

`function<double(double)>`创建六个包装器，用于表示6个函数、函数符和lambda。这样，在对`use_f()`的全部6次调用中，让`F`的类型都相同(`function<double(double)>`)，因此只实例化一次。据此修改后的程序如程序清单18.8所示。

程序清单18.8 wrapped.cpp

```
//wrapped.cpp -- using a function wrapper as an argument
#include "somedefs.h"
#include <iostream>
#include <functional>

double dub(double x) {return 2.0*x;}
double square(double x) {return x*x;}

int main()
{
    using std::cout;
    using std::endl;
    using std::function;

    double y = 1.21;
    function<double(double)> ef1 = dub;
    function<double(double)> ef2 = square;
    function<double(double)> ef3 = Fq(10.0);
    function<double(double)> ef4 = Fp(10.0);
    function<double(double)> ef5 = [] (double u) {return u*u;};
    function<double(double)> ef6 = [] (double u) {return u+u/2.0;};
    cout << "Function pointer dub:\n";
    cout << " " << use_f(y, ef1) << endl;
    cout << "Function pointer square:\n";
    cout << " " << use_f(y, ef2) << endl;
    cout << "Function object Fp:\n";
    cout << " " << use_f(y, ef3) << endl;
    cout << "Function object Fq:\n";
    cout << " " << use_f(y, ef4) << endl;
    cout << "Lambda expression 1:\n";
    cout << " " << use_f(y, ef5) << endl;
    cout << "Lambda expression 2:\n";
    cout << " " << use_f(y,ef6) << endl;
    return 0;
}
```

下面是该程序的示例输出：

```
Function pointer dub:  
use_f count = 1, &count = 0x404020  
2.42  
Function pointer sqrt:  
use_f count = 2, &count = 0x404020  
1.1  
Function object Fp:  
use_f count = 3, &count = 0x404020  
11.21  
Function object Fq:  
use_f count = 4, &count = 0x404020  
12.1  
Lambda expression 1:  
use_f count = 5, &count = 0x404020  
1.4641  
Lambda expression 2:  
use_f count = 6, &count = 0x404020  
1.815
```

从上述输出可知，count的地址都相同，而count的值表明，use_f()被调用了6次。这表明只有一个实例，并调用了该实例6次，这缩小了可执行代码的规模。

18.5.3 其他方式

下面介绍使用function可完成的其他两项任务。首先，在程序清单18.8中，不用声明6个function<double (double)>对象，而只使用一个临时function<double (double)>对象，将其用作函数use_f()的参数：

```

typedef function<double(double)> fdd; // simplify the type declaration
cout << use_f(y, fdd(dub)) << endl; // create and initialize object to dub
cout << use_f(y, fdd(square)) << endl;
...

```

其次，程序清单18.8让use_f()的第二个实参与形参f匹配，但另一种方法是让形参f的类型与原始实参匹配。为此，可在模板use_f()的定义中，将第二个参数声明为function包装器对象，如下所示：

```

#include <functional>
template <typename T>
T use_f(T v, std::function<T(T)> f) // f call signature is T(T)
{
    static int count = 0;
    count++;
    std::cout << " use_f count = " << count
        << ", &count = " << &count << std::endl;
    return f(v);
}

```

这样函数调用将如下：

```

cout << " " << use_f<double>(y, dub) << endl;
...
cout << " " << use_f<double>(y, Fp(5.0)) << endl;
...
cout << " " << use_f<double>(y, [](double u) {return u*u;}) << endl;

```

参数dub、Fp(5.0)等本身的类型并不是function<double(double)>，因此在use_f后面使用了<double>来指出所需的具体化。这样，T被设置为double，而std::function<T(T)>变成了std::function<double(double)>。

18.6 可变参数模板

可变参数模板（variadic template）让您能够创建这样的模板函数和模板类，即可接受可变数量的参数。这里介绍可变参数模板函数。例如，假设要编写一个函数，它可接受任意数量的参数，参数的类型只需

是cout能够显示的即可，并将参数显示为用逗号分隔的列表。请看下面的代码：

```
int n = 14;
double x = 2.71828;
std::string mr = "Mr. String objects!";
show_list(n, x);
show_list(x*x, '!', 7, mr);
```

这里的目标是，定义show_list()，让上述代码能够通过编译并生成如下输出：

```
14, 2.71828
7.38905, !, 7, Mr. String objects!
```

要创建可变参数模板，需要理解几个要点：

- 模板参数包（parameter pack）；
- 函数参数包；
- 展开（unpack）参数包；
- 递归。

18.6.1 模板和函数参数包

为理解参数包的工作原理，首先来看一个简单的模板函数，它显示一个只有一项的列表：

```
template<typename T>
void show_list0(T value)
{
    std::cout << value << ", ";
}
```

在上述定义中，有两个参数列表。模板参数列表只包含T，而函数

参数列表只包含value。下面的函数调用将模板参数列表中的T设置为double，将函数参数列表中的value设置为2.15：

```
show_list0(2.15);
```

C++11提供了一个用省略号表示的元运算符（meta-operator），让您能够声明表示模板参数包的标识符，模板参数包基本上是一个类型列表。同样，它还让您能够声明表示函数参数包的标识符，而函数参数包基本上是一个值列表。其语法如下：

```
template<typename... Args>      // Args is a template parameter pack
void show_list1(Args... args) // args is a function parameter pack
{
...
}
```

其中，Args是一个模板参数包，而args是一个函数参数包。与其他参数名一样，可将这些参数包的名称指定为任何符合C++标识符规则的名称。Args和T的差别在于，T与一种类型匹配，而Args与任意数量（包括零）的类型匹配。请看下面的函数调用：

```
show_list1('S', 80, "sweet", 4.5);
```

在这种情况下，参数包Args包含与函数调用中的参数匹配的类型：char、int、const char *和double。

下面的代码指出value的类型为T：

```
void show_list0(T value)
```

同样，下面的代码指出args的类型为Args：

```
void show_list1(Args... args) // args is a function parameter pack
```

更准确地说，这意味着函数参数包args包含的值列表与模板参数包Args包含的类型列表匹配—无论是类型还是数量。在上面的示例中，args包含值‘S’、80、“sweet”和4.5。

这样，可变参数模板show_list1()与下面的函数调用都匹配：

```
show_list1();
show_list1(99);
show_list1(88.5, "cat");
show_list1(2, 4, 6, 8, "who do we", std::string("appreciate));
```

就最后一个函数调用而言，模板参数包Args包含类型int、int、int、int、const char *和std::string，而函数参数包args包含值2、4、6、8、“who do we”和std::string(“appreciate”)。

18.6.2 展开参数包

但函数如何访问这些包的内容呢？索引功能在这里不适用，即您不能使用Args[2]来访问包中的第三个类型。相反，可将省略号放在函数参数包名的右边，将参数包展开。例如，请看下述有缺陷的代码：

```
template<typename... Args> // Args is a template parameter pack
void show_list1(Args... args) // args is a function parameter pack
{
    show_list1(args...); // passes unpacked args to show_list1()
}
```

这是什么意思呢？为何说它存在缺陷？假设有如下函数调用：

```
show_list1(5, 'L', 0.5);
```

这将把5、‘L’和0.5封装到args中。在该函数内部，下面的调用：

```
show_list1(args...);
```

将展开成如下所示：

```
show_list1(5, 'L', 0.5);
```

也就是说，args被替换为三给存储在args中的值。因此，表示法args...展开为一个函数参数列表。不幸的是，该函数调用与原始函数调用相同，因此它将使用相同的参数不断调用自己，导致无限递归（这存在缺陷）。

18.6.3 在可变参数模板函数中使用递归

虽然前面的递归让show_list1()成为有用函数的希望破灭，但正确使用递归为访问参数包的内容提供了解决方案。这里的核理念是，将函数参数包展开，对列表中的第一项进行处理，再将余下的内容传递给递归调用，以此类推，直到列表为空。与常规递归一样，确保递归将终止很重要。这里的技巧是将模板头改为如下所示：

```
template<typename T, typename... Args>
void show_list3( T value, Args... args)
```

对于上述定义，show_list3()的第一个实参决定了T和value的值，而其他实参决定了Args和args的值。这让函数能够对value进行处理，如显示它。然后，可递归调用show_list3()，并以args...的方式将其他实参传递给它。每次递归调用都将显示一个值，并传递缩短了的列表，直到列表为空为止。程序清单18.9提供了一种实现，它虽然不完美，但演示了这种技巧。

程序清单18.9 variadic1.cpp

```
//variadic1.cpp -- using recursion to unpack a parameter pack
#include <iostream>
#include <string>
```

```
// definition for 0 parameters -- terminating call
void show_list3() {}

// definition for 1 or more parameters
template<typename T, typename... Args>
void show_list3( T value, Args... args)
{
    std::cout << value << ", ";
    show_list3(args...);
}

int main()
{
    int n = 14;
    double x = 2.71828;
    std::string mr = "Mr. String objects!";
    show_list3(n, x);
    show_list3(x*x, '!', 7, mr);
    return 0;
}
```

1. 程序说明

请看下面的函数调用：

```
show_list3(x*x, '!', 7, mr);
```

第一个实参导致T为double, value为x*x。其他三种类型（char、int和std::string）将放入Args包中，而其他三个值（'!'、7和mr）将放入args包中。

接下来，函数show_list3()使用cout显示value（大约为7.38905）和字符串“,”。这完成了显示列表中第一项的工作。

接下来是下面的调用：

```
show_list3(args...);
```

考虑到args...的展开作用，这与如下代码等价：

```
show_list3('!', 7, mr);
```

前面说过，列表将每次减少一项。这次T和value分别为char和‘!’，而余下的两种类型和两个值分别被包装到Args和args中，下次递归调用将处理这些缩小了的包。最后，当args为空时，将调用不接受任何参数的show_list3()，导致处理结束。

程序清单18.9中两个函数调用的输出如下：

```
14, 2.71828, 7.38905, !, 7, Mr. String objects!,
```

2. 改进

可对show_list3()做两方面的改进。当前，该函数在列表的每项后面显示一个逗号，但如果能省去最后一项后面的逗号就好了。为此，可添加一个处理一项的模板，并让其行为与通用模板稍有不同：

```
// definition for 1 parameter
template<typename T>
void show_list3(T value)
{
    std::cout << value << '\n';
}
```

这样，当args包缩短到只有一项时，将调用这个版本，而它打印换行符而不是逗号。另外，由于没有递归调用show_list3()，它也将终止递归。

另一个可改进的地方是，当前的版本按值传递一切。对于这里使用的简单类型来说，这没问题，但对于cout可打印的大型类来说，这样做的效率很低。在可变参数模板中，可指定展开模式（pattern）。为此，可将下述代码：

```
show_list3(Args... args);
```

替换为如下代码：

```
show_list3(const Args&... args);
```

这将对每个函数参数应用模式const &。这样，最后分析的参数将不是std::string mr，而是const std::string& mr。

程序清单18.10包含这两项修改。

程序清单18.10 variadic2.cpp

```
// variadic2.cpp
#include <iostream>
#include <string>

// definition for 0 parameters
void show_list() {}

// definition for 1 parameter
template<typename T>
void show_list(const T& value)
{
    std::cout << value << '\n';
}
```

```
// definition for 2 or more parameters
template<typename T, typename... Args>
void show_list(const T& value, const Args&... args)
{
    std::cout << value << ", ";
    show_list(args...);
}

int main()
{
    int n = 14;
    double x = 2.71828;
    std::string mr = "Mr. String objects!";
    show_list(n, x);
    show_list(x*x, '!', 7, mr);
    return 0;
}
```

该程序的输出如下：

```
14, 2.71828
7.38905, !, 7, Mr. String objects!
```

18.7 C++11新增的其他功能

C++11增加了很多功能，本书无法全面介绍；另外，本书编写期间，其中很多功能还未得到广泛实现。然而，有些功能有必要简要地介绍一下。

18.7.1 并行编程

当前，为提高计算机性能，增加处理器数量比提高处理器速度更容易。因此，装备了双核、四核处理器甚至多个多核处理器的计算机很常见，这让计算机能够同时执行多个线程，其中一个处理器可能处理视频下载，而另一个处理器处理电子表格。

有些操作能受益于多线程，但有些不能。考虑单向链表的搜索：程序必须从链表开头开始，沿链接依次向下搜索，直到到达链表末尾；在这种情况下，多线程的帮助不大。再来看未经排序的数组。考虑到数组的随机存取特征，可让一个线程从数组开头开始搜索，并让另一个线程从数组中间开始搜索，这将让搜索时间减半。

多线程确实带来了很多问题。如果一个线程挂起或两个线程试图同时访问同一项数据，结果将如何呢？为解决并行性问题，C++定义了一个支持线程化执行的内存模型，添加了关键字`thread_local`，提供了相关的库支持。关键字`thread_local`将变量声明为静态存储，其持续性与特定线程相关；即定义这种变量的线程过期时，变量也将过期。

库支持由原子操作（atomic operation）库和线程支持库组成，其中原子操作库提供了头文件`atomic`，而线程支持库提供了头文件`thread`、`mutex`、`condition_variable`和`future`。

18.7.2 新增的库

C++11添加了多个专用库。头文件`random`支持的可扩展随机数库提供了大量比`rand()`复杂的随机数工具。例如，您可以选择随机数生成器和分布状态，分布状态包括均匀分布（类似于`rand()`）、二项式分布和正态分布等。

头文件`chrono`提供了处理时间间隔的途径。

头文件`tuple`支持模板`tuple`。`tuple`对象是广义的`pair`对象。`pair`对象可存储两个类型不同的值，而`tuple`对象可存储任意多个类型不同的值。

头文件`ratio`支持的编译阶段有理数算术库让您能够准确地表示任何有理数，其分子和分母可用最宽的整型表示。它还支持对这些有理数进行算术运算。

在新增的库中，最有趣的一个是头文件`regex`支持的正则表达式

库。正则表达式指定了一种模式，可用于与文本字符串的内容匹配。例如，方括号表达式与方括号中的任何单个字符匹配，因此[cCkK]与c、C、k和K都匹配，而[cCkK] at与单词cat、Cat、kat和Kat都匹配。其他模式包括与一位数字匹配的\d、与一个单词匹配的\w、与制表符匹配的\t等。在C++中，斜杠具有特殊含义，因此对于模式\d\t\w\d（即依次为一位数字、制表符、单词和一位数字），必须写成字符串面量“\d\t\w\d”，即使用\表示\。这是引入原始字符串的原因之一（参见第4章），它让您能够将该模式写成R“\d\t\w\d”。

ed、grep和awk等UNIX工具都使用正则表达式，而解释型语言Perl扩展了正则表达式的功能。C++正则表达式库让您能够选择多种形式的正则表达式。

18.7.3 低级编程

低级编程中的“低级”指的是抽象程度，而不是编程质量。低级意味着接近于计算机硬件和机器语言使用的比特和字节。对嵌入式编程和改善操作的效率而言，低级编程很重要。C++11给低级编程人员提供了一些帮助。

变化之一是放松了POD（Plain Old Data）的要求。在C++98中，POD是标量类型（单值类型，如int或double）或没有构造函数、基类、私有数据、虚函数等的老式结构。以前的理念是，POD是可安全地逐字节复制的东西。这种理念没变，但C++11认识到，在满足C++98的某些约束的情况下，仍可以是合法的POD。这有助于低级编程，因为有些低级操作（如使用C语言函数进行逐字节复制或二进制I/O）要求处理对象为POD。

另一项修改是，允许共用体的成员有构造函数和析构函数，这让共用体更灵活；但保留了其他一些限制，如成员不能有虚函数。在需要最大程度地减少占用的内存时，通常使用共用体；上述新规则在这些情况下给程序员有更大的灵活性和功能。

C++11解决了内存对齐问题。计算机系统可能对数据在内存中的存储方式有一定的限制。例如，一个系统可能要求double值的内存地址为偶数，而另一个系统可能要求其起始位置为8的整数倍。要获悉有关类型或对象的对齐要求，可使用运算符alignof()（参见附录E）。要控制对齐方式，可使用说明符alignas。

`constexpr`机制让编译器能够在编译阶段计算结果为常量的表达式，让`const`变量可存储在只读内存中，这对嵌入式编程来说很有用（在运行阶段初始化的变量存储在随机访问内存中）。

18.7.4 杂项

C99引入了依赖于实现的扩展整型，C++11继承了这种传统。在使用128位整数的系统中，可使用这样的类型。在C语言中，扩展类型由头文件`stdint.h`支持，而在C++中，为头文件`cstdint`。

C++11提供了一种创建用户自定义字面量的机制：字面量运算符（literal operator）。使用这种机制可定义二进制字面量，如`1001001b`，相应的字面量运算符将把它转换为整数值。

C++提供了调试工具`assert`。这是一个宏，它在运行阶段对断言进行检查，如果为`true`，则显示一条消息，否则调用`abort()`。断言通常是程序员认为在程序的某个阶段应为`true`的东西。C++11新增了关键字`static_assert`，可用于在编译阶段对断言进行测试。这样做的主要目的在于，对于在编译阶段（而不是运行阶段）实例化的模板，调试起来将更简单。

C++11加强了对元编程（metaprogramming）的支持。元编程指的是编写这样的程序，它创建或修改其他程序，甚至修改自身。在C++中，可使用模板在编译阶段完成这种工作。

18.8 语言变化

计算机语言是如何成长和发展的呢？C++的使用范围足够广后，显然需要国际标准，并将其控制权交给标准委员会：最初是ANSI委员会，随后是ISO/ANSI联合委员会，当前是ISO/IEC JTC1/SC22/WG21（C++标准委员会）。ISO是国际标准组织，IEC是国际电子技术委员会，JEC1是前两家组织组建的联合技术委员会1，SC22是JTC1下属的编程语言委员会，而WG21是SC22下属的C++工作小组。

委员会考虑缺陷报告和有关语言修改和扩展的提议，并试图达成一致。这个过程既繁琐又漫长，《The Design and Evolution of C++》（Stroustrup, Addison-Wesley, 1994）介绍了这方面的一些情况。寻求

一致的委员会沉闷而争议不断，可能不是鼓励创新的好方式，这也不是标准委员会应扮演的角色。

但就C++而言，还有另一种变更的途径，那就是充满创意的C++编程社区的直接行动。程序员无法不受羁绊地改进语言，但可创建有用的库。设计良好的库可改善语言的用途和功能，提高可靠性，让编程更容易、更有乐趣。库是在现有语言功能的基础上创建的，不需要额外的编译器支持。如果库是通过模板实现的，则可以头文件（文本文件）的方式分发。

一项这样的变革是STL，它主要是Alexander Stepanov创建的，Hewlett-Packard免费提供它。STL在编程社区获得了巨大成功，成了第一个ANSI/ISO标准的候选内容。事实上，其设计影响新标准的其他方面。

18.8.1 Boost项目

最近，Boost库成了C++编程的重要部分，给C++11带来了深远影响。Boost项目发起于1998年，当时的C++库工作小组主席Beman Dawes召集其他几位小组成员制定了一项计划，准备在标准委员会的框架外创建新库。该计划的基本理念是，创建一个充当开放论坛的网站，让人发布免费的C++库。这个项目提供有关许可和编程实践的指南，并要求对提议的库进行同行审阅。其最终的成果是，一系列得到高度赞扬和广泛使用的库。这个项目提供了一个环境，让编程社区能够检验和评估编程理念以及提供反馈。

18.8.2 TR1

TR1（Technical Report 1）是C++标准委员会的部分成员发起的一个项目，它是一个库扩展选集，这些扩展与C++98标准兼容，但不是必不可少的。这些扩展是下一个C++标准的候选内容。TR1库让C++社区能够检验其组成部分的价值。当标准委员会将TR1的大部分内容融入C++11时，面对的是众所皆知且经过实践检验的库。

在TR1中，Boost库占了很大一部分。这包括模板类tuple和array、模板bind和function、智能指针（对名称和实现做了一定的修改）、static_assert、regex库和random库。另外，Boost社区和TR1用户的经

也导致了实际的语言变更，如异常规范的摒弃和可变参数模板的添加，其中可变参数模板让tuple模板类和function模板的实现更好了。

18.8.3 使用Boost

虽然在C++11中，可访问Boost开发的众多库，但还有很多其他的Boost库。例如，Conversion库中的lexical_cast让您能够在数值和字符串类型之间进行简单地转换，其语法类似于dynamic_cast：将模板参数指定为目标类型。程序清单18.11是一个简单示例。

程序清单18.11 lexcast.cpp

```
// lexcast.cpp -- simple cast from float to string
#include <iostream>
#include <string>
#include "boost/lexical_cast.hpp"
int main()
{
    using namespace std;
    cout << "Enter your weight: ";
    float weight;
    cin >> weight;
    string gain = "A 10% increase raises ";
    string wt = boost::lexical_cast<string>(weight);
    gain = gain + wt + " to "; // string operator+()
    weight = 1.1 * weight;
    gain = gain + boost::lexical_cast<string>(weight) + ".";
    cout << gain << endl;
    return 0;
}
```

下面是两次运行该程序的情况：

```
Enter your weight: 150
A 10% increase raises 150 to 165.
```

```
Enter your weight: 156
A 10% increase raises 156 to 171.600006.
```

第二次运行的结果凸显了lexical_cast的局限性：它未能很好地控制浮点数的格式。为控制浮点数的格式，需要使用更精致的内核格式化工具，这在第17章讨论过。

还可以使用lexical_cast将字符串转换为数值。

显然，Boost提供的功能比这里介绍的要多得多。例如，Any库让您能够在STL容器中存储一系列不同类型的值和对象，方法是将Any模板用作各种值的包装器。Math库在标准math库的基础上增加了数学函数。Filesystem库让您编写的代码可在使用不同文件系统的平台之间移植。有关这个库以及如何将其加入到各种平台的更详细信息，请参阅Boost网站（www.boost.org）。另外，有些C++编译器（如Cygwin编译器）还自带了Boost库。

18.9 接下来的任务

如果仔细阅读了本书，则应很好地掌握了C++的规则。然而，这仅仅是学习这种语言的开始，接下来需要学习如何高效地使用该语言，这样的路更长。最好的情况是，工作或学习环境让您能够接触优秀的C++代码和程序员。另外，了解C++后，便可以阅读一些介绍高级主题和面向对象编程的书籍，附录H列出了一些这样的资源。

OOP有助于开发大型项目，并提高其可靠性。OOP方法的基本活动之一是发明能够表示正在模拟的情形（被称为问题域（problem domain））的类。由于实际问题通常很复杂，因此找到适当的类富有挑战性。创建复杂的系统时，从空白开始通常不可行，最好采用逐步迭代的方式。为此，该领域的实践者开发了多种技术和策略。具体地说，重要的是在分析和设计阶段完成尽可能多的迭代工作，而不要不断地修改实际代码。

常用的技术有两种：用例分析（use-case analysis）和CRC卡（CRC card）。在用例分析中，开发小组列出了常见的使用方式或最终系统将用于的场景；找出元素、操作和职责，以确定可能要使用的类和类特性。CRC（Class/Responsibilities/Collaborators的简称）卡片是一种分析场景的简单方法。开发小组为每个类创建索引卡片，卡片上列出了类名、类责任（如表示的数据和执行的操作）以及类的协作者（如必须与之交互的其他类）。然后，小组使用CRC卡片提供的接口模拟场景。这可能提出新的类、转换责任等。

在更大的规模上，是用于整个项目的系统方法。在这方面，最新的工具是统一建模语言（Unified Modeling Language, UML），它不是一种编程语言，而是一种用于表示编程项目的分析和设计语言，是由Grady Booch、Jim Rumbaugh和Ivar Jacobson开发的，他们分别是更早的3种建模语言（Booch Method、OMT（对象建模技术，Object Modeling Technique）和OOSE（面向对象的软件工程，Object-Oriented Software Engineering））的主要开发人员。UML是从这3种语言演化而来的，于2005年被ISO/IEC批准为标准。

除加深对C++的总体理解外，还可能需要学习特定的类库。例如，Microsoft和Embarcadero提供了大量简化Windows编程的类库，而Apple Xcode提供了简化Apple平台（如iPhone）编程的类库。

18.10 总结

C++新标准新增了大量功能。有些旨在让C++更容易学习和使用，这包括用大括号括起的统一的列表初始化、使用auto自动推断类型、类内成员初始化以及基于范围的for循环；而有些旨在增强类设计以及使其更容易理解，这包括默认的和禁用的方法、委托构造函数、继承构造函数以及让虚函数设计更清晰的说明符override和final。

有几项改进旨在提供程序和编程效率。lambda表达式比函数指针和函数符更好，模板function可用于减少模板实例数量，右值引用让您能够使用移动语义以及实现移动构造函数和移动赋值运算符。

其他改进提供了更佳的工作方式。作用域内枚举让您能够更好地控制枚举的作用域和底层类型；模板unique_ptr和shared_ptr让您能够更好地处理使用new分配的内存。

新增的`decltype`、返回类型后置、模板别名和可变参数模板让模板设计得到了改进。

修改后的共用体和POD规则、`alignof()`运算符、`alignas`说明符以及`constexpr`机制支持低级编程。

新增了多个库（包括新的STL类、`tuple`模板和`regex`库）为众多常见的编程问题提供了解决方案。

为支持并行编程，新标准还添加了关键字`thread_local`和`atomic`库。

总之，无论对新手还是专家来说，新标准都改善了C++的可用性和可靠性。

18.11 复习题

1. 使用用大括号括起的初始化列表语法重写下述代码。重写后的代码不应使用数组`ar`:

```
class Z200
{
private:
    int j;
    char ch;
    double z;
public:
    Z200(int jv, char chv, zv) : j(jv), ch(chv), z(zv) {}
    ...
};

double x = 8.8;
std::string s = "What a bracing effect!";
int k(99);
Z200 zip(200, 'Z', 0.675);
std::vector<int> ai(5);
int ar[5] = {3, 9, 4, 7, 1};
for (auto pt = ai.begin(), int i = 0; pt != ai.end(); ++pt, ++i)
    *pt = ai[i];
```

2. 在下述简短的程序中，哪些函数调用不对？为什么？对于合法的函数调用，指出其引用参数指向的是什么。

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

double up(double x) { return 2.0*x; }
void r1(const double &rx) {cout << rx << endl;}
void r2(double &rx) {cout << rx << endl;}
void r3(double &&rx) {cout << rx << endl; }

int main()
{
    double w = 10.0;
    r1(w);
    r1(w+1);
    r1(up(w));
    r2(w);
    r2(w+1);
    r2(up(w));
    r3(w);
    r3(w+1);
    r3(up(w));
    return 0;
}
```

3. a. 下述简短的程序显示什么？为什么？

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

double up(double x) { return 2.0* x; }
void r1(const double &rx) {cout << "const double & rx\n"; }
void r1(double &rx) {cout << "double & rx\n"; }

int main()
{
    double w = 10.0;
    r1(w);
    r1(w+1);
    r1(up(w));
    return 0;
}
```

b. 下述简短的程序显示什么？为什么？

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

double up(double x) { return 2.0*x; }
void r1(double &rx) {cout << "double & rx\n"; }
void r1(double &&rx) {cout << "double && rx\n"; }

int main()
{
    double w = 10.0;
    r1(w);
    r1(w+1);
    r1(up(w));
    return 0;
}
```

c. 下述简短的程序显示什么？为什么？

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

```
double up(double x) {return 2.0*x;}
void r1(const double &rx) {cout << "const double & rx\n";}
void r1(double &&rx) {cout << "double && rx\n";}

int main()
{
    double w = 10.0;
    r1(w);
    r1(w+1);
    r1(up(w));
    return 0;
}
```

4. 哪些成员函数是特殊的成员函数？它们特殊的原因是什么？

5. 假设Fizzle类只有如下所示的数据成员：

```
class Fizzle
{
private:
    double bubbles[4000];
...
};
```

为什么不适合给这个类定义移动构造函数？要让这个类适合定义移动构造函数，应如何修改存储4000个double值的方式？

6. 修改下述简短的程序，使其使用lambda表达式而不是f1()。请不要修改show2()。

```
#include <iostream>
template<typename T>
void show2(double x, T& fp) { std::cout << x << " -> " << fp(x) << '\n'; }
double f1(double x) { return 1.8*x + 32; }
int main()
{
    show2(18.0, f1);
    return 0;
}
```

7. 修改下述简短而丑陋的程序，使其使用lambda表达式而不是函数符Adder。请不要修改sum()。

```
#include <iostream>
#include <array>
const int Size = 5;
template<typename T>
```

```

void sum(std::array<double,Size> a, T& fp);
class Adder
{
    double tot;
public:
    Adder(double q = 0) : tot(q) {}
    void operator()(double w) { tot +=w; }
    double tot_v () const {return tot;}
};
int main()
{
    double total = 0.0;
    Adder ad(total);
    std::array<double, Size> temp_c = {32.1, 34.3, 37.8, 35.2, 34.7};
    sum(temp_c,ad);
    total = ad.tot_v();
    std::cout << "total: " << ad.tot_v() << '\n';
    return 0;
}
template<typename T>
void sum(std::array<double,Size> a, T& fp)
{
    for(auto pt = a.begin(); pt != a.end(); ++pt)
    {
        fp(*pt);
    }
}

```

18.12 编程练习

1. 下面是一个简短程序的一部分：

```
int main()
{
    using namespace std;
    // list of double deduced from list contents
    auto q = average_list({15.4, 10.7, 9.0});
    cout << q << endl;
    // list of int deduced from list contents
    cout << average_list({20, 30, 19, 17, 45, 38}) << endl;
    // forced list of double
    auto ad = average_list<double>({'A', 70, 65.33});
    cout << ad << endl;
    return 0;
}
```

请提供函数average_list()，让该程序变得完整。它应该是一个模板函数，其中的类型参数指定了用作函数参数的initilize_list模板的类型以及函数的返回类型。

2. 下面是类Cpmv的声明：

```

class Cpmv
{
public:
    struct Info
    {
        std::string qcode;
        std::string zcode;
    };
private:
    Info *pi;
public:
    Cpmv();
    Cpmv(std::string q, std::string z);
    Cpmv(const Cpmv & cp);
    Cpmv(Cpmv && mv);
    ~Cpmv();
    Cpmv & operator=(const Cpmv & cp);
    Cpmv & operator=(Cpmv && mv);
    Cpmv operator+(const Cpmv & obj) const;
    void Display() const;
};

```

函数operator+()应创建一个对象，其成员qcode和zcode有操作数的相应成员拼接而成。请提供为移动构造函数和移动赋值运算符实现移动语义的代码。编写一个使用所有这些方法的程序。为方便测试，让各个方法都显示特定的内容，以便知道它们被调用。

3. 编写并测试可变参数模板函数sum_value(), 它接受任意长度的

参数列表（其中包含数值，但可以是任何类型），并以long double的方式返回这些数值的和。

4. 使用lambda重新编写程序清单16.5。具体地说，使用一个有名称的lambda替换函数outint()，并将函数符替换为两个匿名lambda表达式。