第 15 章　友元、异常和其他

本章内容包括：

- 友元类。
- 友元类方法。
- 嵌套类。
- 引发异常、try 块和 catch 块。
- 异常类。
- 运行阶段类型识别（RTTI）。
- dynamic_cast 和 typeid。
- static_cast、const_cast 和 reiterpret_cast。

本章先介绍一些 C++语言最初就有的特性，然后介绍 C++语言新增的一些特性。前者包括友元类、友元成员函数和嵌套类，它们是在其他类中声明的类；后者包括异常、运行阶段类型识别（RTTI）和改进后的类型转换控制。C++异常处理提供了处理特殊情况的机制，如果不对其进行处理，将导致程序终止。RTTI 是一种确定对象类型的机制。新的类型转换运算符提高了类型转换的安全性。后 3 种特性是 C++新增的，老式编译器不支持它们。

15.1　友元

本书前面的一些示例将友元函数用于类的扩展接口中，类并非只能拥有友元函数，也可以将类作为友元。在这种情况下，友元类的所有方法都可以访问原始类的私有成员和保护成员。另外，也可以做更严格的限制，只将特定的成员函数指定为另一个类的友元。哪些函数、成员函数或类为友元是由类定义的，而不能从外部强加友情。因此，尽管友元被授予从外部访问类的私有部分的权限，但它们并不与面向对象的编程思想相悖；相反，它们提高了公有接口的灵活性。

15.1.1　友元类

什么时候希望一个类成为另一个类的友元呢？我们来看一个例子。假定需要编写一个模拟电视机和遥控器的简单程序。决定定义一个 Tv 类和一个 Remote 类，来分别表示电视机和遥控器。很明显，这两个类之间应当存在某种关系，但是什么样的关系呢？遥控器并非电视机，反之亦然，所以公有继承的 is-a 关系并不适用。遥控器也非电视机的一部分，反之亦然，因此包含或私有继承和保护继承的 has-a 关系也不适用。事实上，遥控器可以改变电视机的状态，这表明应将 Romote 类作为 Tv 类的一个友元。

首先定义 Tv 类。可以用一组状态成员（描述电视各个方面的变量）来表示电视机。下面是一些可能的状态：

- 开/关；
- 频道设置；

- 音量设置；
- 有线电视或天线调节模式；
- TV 调谐或 A/V 输入。

调节模式指的是，在美国，对于有线接收和 UHF 广播接收，14 频道和 14 频道以上的频道间隔是不同的。输入选择包括 TV（有线 TV 或广播 TV）和 DVD。有些电视机可能提供更多的选择，如多种 DVD/蓝光输入，但对于这个示例的目的而言，这个清单足够了。

另外，电视机还有一些不是状态变量的参数。例如，可接收频道数随电视机而异，可以包括一个记录这个值的成员。

接下来，必须给类提供一些修改这些设置的方法。当前，很多电视机都将控件藏在面板后面，但大多数电视机还是可以在不使用遥控器的情况下进行换台等工作的，通常只能逐频道换台，而不能随意选台。同样，通常还有两个按钮，分别用来增加和降低音量。

遥控器的控制能力应与电视机内置的控制功能相同，它的很多方法都可通过使用 Tv 方法来实现。另外，遥控器通常都提供随意选择频道的功能，即可以直接从 2 频道换到 20 频道，并不用逐次切换频道。另外，很多遥控器都有多种工作模式，如用作电视控制器和 DVD 遥控器。

这些考虑因素表明，定义应类似于程序清单 15.1。定义中包括一些被定义为枚举的常数。下面的语句使 Remote 成为友元类：

```
friend class Remote;
```

友元声明可以位于公有、私有或保护部分，其所在的位置无关紧要。由于 Remote 类提到了 Tv 类，所以编译器必须了解 Tv 类后，才能处理 Remote 类，为此，最简单的方法是首先定义 Tv 类。也可以使用前向声明（forward delaration），这将稍后介绍。

程序清单 15.1　tv.h

```
// tv.h -- Tv and Remote classes
#ifndef TV_H_
#define TV_H_

class Tv
{
public:
    friend class Remote;   // Remote can access Tv private parts
    enum {Off, On};
    enum {MinVal,MaxVal = 20};
    enum {Antenna, Cable};
    enum {TV, DVD};

    Tv(int s = Off, int mc = 125) : state(s), volume(5),
        maxchannel(mc), channel(2), mode(Cable), input(TV) {}
    void onoff() {state = (state == On)? Off : On;}
    bool ison() const {return state == On;}
    bool volup();
    bool voldown();
    void chanup();
    void chandown();
    void set_mode() {mode = (mode == Antenna)? Cable : Antenna;}
    void set_input() {input = (input == TV)? DVD : TV;}
    void settings() const; // display all settings
private:
    int state;             // on or off
    int volume;            // assumed to be digitized
    int maxchannel;        // maximum number of channels
```

```
    int channel;            // current channel setting
    int mode;               // broadcast or cable
    int input;              // TV or DVD
};
class Remote
{
private:
    int mode;               // controls TV or DVD
public:
    Remote(int m = Tv::TV) : mode(m) {}
    bool volup(Tv & t) { return t.volup();}
    bool voldown(Tv & t) { return t.voldown();}
    void onoff(Tv & t) { t.onoff(); }
    void chanup(Tv & t) {t.chanup();}
    void chandown(Tv & t) {t.chandown();}
    void set_chan(Tv & t, int c) {t.channel = c;}
    void set_mode(Tv & t) {t.set_mode();}
    void set_input(Tv & t) {t.set_input();} .
};
 #endif
```

在程序清单 15.1 中，大多数类方法都被定义为内联的。除构造函数外，所有的 Romote 方法都将一个 Tv 对象引用作为参数，这表明遥控器必须针对特定的电视机。程序清单 15.2 列出了其余的定义。音量设置函数将音量成员增减一个单位，除非声音到达最大或最小。频道选择函数使用循环方式，最低的频道设置为 1，它位于最高的频道设置 maxchannel 之后。

很多方法都使用条件运算符在两种状态之间切换：

```
void onoff() {state = (state == On)? Off : On;}
```

如果两种状态值分别为 true（1）和 false（0），则可以结合使用将在附录 E 讨论的按位异或和赋值运算符（^=）来简化上述代码：

```
void onoff() {state ^= 1;}
```

事实上，在单个无符号 char 变量中可存储多达 8 个双状态设置，分别对它们进行切换；但现在已经不用这样做了，使用附录 E 中讨论的按位运算符就可以完成。

程序清单 15.2　tv.cpp

```
// tv.cpp -- methods for the Tv class (Remote methods are inline)
#include <iostream>
#include "tv.h"

bool Tv::volup()
{
    if (volume < MaxVal)
    {
        volume++;
        return true;
    }
    else
        return false;
}
bool Tv::voldown()
{
    if (volume > MinVal)
    {
```

```
        volume--;
        return true;
    }
    else
        return false;
}

void Tv::chanup()
{
    if (channel < maxchannel)
        channel++;
    else
        channel = 1;
}

void Tv::chandown()
{
    if (channel > 1)
        channel--;
    else
        channel = maxchannel;
}

void Tv::settings() const
{
    using std::cout;
    using std::endl;
    cout << "TV is " << (state == Off? "Off" : "On") << endl;
    if (state == On)
    {
        cout << "Volume setting = " << volume << endl;
        cout << "Channel setting = " << channel << endl;
        cout << "Mode = "
            << (mode == Antenna? "antenna" : "cable") << endl;
        cout << "Input = "
            << (input == TV? "TV" : "DVD") << endl;
    }
}
```

程序清单 15.3 是一个简短的程序，可以测试一些特性。另外，可使用同一个遥控器控制两台不同的电视机。

程序清单 15.3　use_tv.cpp

```
//use_tv.cpp -- using the Tv and Remote classes
#include <iostream>
#include "tv.h"

int main()
{
    using std::cout;
    Tv s42;
    cout << "Initial settings for 42\" TV:\n";
    s42.settings();
    s42.onoff();
    s42.chanup();
    cout << "\nAdjusted settings for 42\" TV:\n";
```

```
    s42.chanup();
    cout << "\nAdjusted settings for 42\" TV:\n";
    s42.settings();

    Remote grey;

    grey.set_chan(s42, 10);
    grey.volup(s42);
    grey.volup(s42);
    cout << "\n42\" settings after using remote:\n";
    s42.settings();

    Tv s58(Tv::On);
    s58.set_mode();
    grey.set_chan(s58,28);
    cout << "\n58\" settings:\n";
    s58.settings();
    return 0;
}
```

下面是程序清单 15.1～程序清单 15.3 组成的程序的输出：

```
Initial settings for 42" TV:
TV is Off
Adjusted settings for 42" TV:
TV is On
Volume setting = 5
Channel setting = 3
Mode = cable
Input = TV

42" settings after using remote:
TV is On
Volume setting = 7
Channel setting = 10
Mode = cable
Input = TV

58" settings:
TV is On
Volume setting = 5
Channel setting = 28
Mode = antenna
Input = TV
```

这个练习的主要目的在于表明，类友元是一种自然用语，用于表示一些关系。如果不使用某些形式的友元关系，则必须将 Tv 类的私有部分设置为公有的，或者创建一个笨拙的、大型类来包含电视机和遥控器。这种解决方法无法反应这样的事实，即同一个遥控器可用于多台电视机。

15.1.2 友元成员函数

从上一个例子中的代码可知，大多数 Remote 方法都是用 Tv 类的公有接口实现的。这意味着这些方法不是真正需要作为友元。事实上，唯一直接访问 Tv 成员的 Remote 方法是 Remote::set_chan( )，因此它是唯一需要作为友元的方法。确实可以选择仅让特定的类成员成为另一个类的友元，而不必让整个类成为友元，但这样做稍微有点麻烦，必须小心排列各种声明和定义的顺序。下面介绍其中的原因。

让 Remote::set_chan( )成为 Tv 类的友元的方法是，在 Tv 类声明中将其声明为友元：

```
class Tv
{
    friend void Remote::set_chan(Tv & t, int c);
    ...
};
```

然而，要使编译器能够处理这条语句，它必须知道 Remote 的定义。否则，它无法知道 Remote 是一个类，而 set_chan 是这个类的方法。这意味着应将 Remote 的定义放到 Tv 的定义前面。Remote 的方法提到了 Tv 对象，而这意味着 Tv 定义应当位于 Remote 定义之前。避开这种循环依赖的方法是，使用前向声明（forward declaration）。为此，需要在 Remote 定义的前面插入下面的语句：

```
class Tv;               // forward declaration
```

这样，排列次序应如下：

```
class Tv;  // forward declaration
class Remote { ... };
class Tv { ... };
```

能否像下面这样排列呢？

```
class Remote;           // forward declaration
class Tv { ... };
class Remote { ... };
```

答案是不能。原因在于，在编译器在 Tv 类的声明中看到 Remote 的一个方法被声明为 Tv 类的友元之前，应该先看到 Remote 类的声明和 set_chan( )方法的声明。

还有一个麻烦。程序清单 15.1 的 Remote 声明包含了内联代码，例如：

```
void onoff(Tv & t) { t.onoff(); }
```

由于这将调用 Tv 的一个方法，所以编译器此时必须已经看到了 Tv 类的声明，这样才能知道 Tv 有哪些方法，但正如看到的，该声明位于 Remote 声明的后面。这种问题的解决方法是，使 Remote 声明中只包含方法声明，并将实际的定义放在 Tv 类之后。这样，排列顺序将如下：

```
class Tv;                // forward declaration
class Remote { ... };      // Tv-using methods as prototypes only
class Tv { ... };
// put Remote method definitions here
```

Remote 方法的原型与下面类似：

```
void onoff(Tv & t);
```

检查该原型时，所有的编译器都需要知道 Tv 是一个类，而前向声明提供了这样的信息。当编译器到达真正的方法定义时，它已经读取了 Tv 类的声明，并拥有了编译这些方法所需的信息。通过在方法定义中使用 inline 关键字，仍然可以使其成为内联方法。程序清单 15.4 列出了修订后的头文件。

程序清单 15.4　tvfm.h

```
// tvfm.h -- Tv and Remote classes using a friend member
#ifndef TVFM_H_
#define TVFM_H_

class Tv;                        // forward declaration

class Remote
{
public:
    enum State{Off, On};
    enum {MinVal,MaxVal = 20};
    enum {Antenna, Cable};
```

```
    enum {TV, DVD};
private:
    int mode;
public:
    Remote(int m = TV) : mode(m) {}
    bool volup(Tv & t);          // prototype only
    bool voldown(Tv & t);
    void onoff(Tv & t) ;
    void chanup(Tv & t) ;
    void chandown(Tv & t) ;
    void set_mode(Tv & t) ;
    void set_input(Tv & t);
    void set_chan(Tv & t, int c);

};

class Tv
{
public:
    friend void Remote::set_chan(Tv & t, int c);
    enum State{Off, On};
    enum {MinVal,MaxVal = 20};
    enum {Antenna, Cable};
    enum {TV, DVD};

    Tv(int s = Off, int mc = 125) : state(s), volume(5),
        maxchannel(mc), channel(2), mode(Cable), input(TV) {}
    void onoff() {state = (state == On)? Off : On;}
    bool ison() const {return state == On;}
    bool volup();
    bool voldown();
    void chanup();
    void chandown();
    void set_mode() {mode = (mode == Antenna)? Cable : Antenna;}
    void set_input() {input = (input == TV)? DVD : TV;}
    void settings() const;
private:
    int state;
    int volume;
    int maxchannel;
    int channel;
    int mode;
    int input;
};

// Remote methods as inline functions
inline bool Remote::volup(Tv & t) { return t.volup();}
inline bool Remote::voldown(Tv & t) { return t.voldown();}
inline void Remote::onoff(Tv & t) { t.onoff(); }
inline void Remote::chanup(Tv & t) {t.chanup();}
inline void Remote::chandown(Tv & t) {t.chandown();}
inline void Remote::set_mode(Tv & t) {t.set_mode();}
inline void Remote::set_input(Tv & t) {t.set_input();}
inline void Remote::set_chan(Tv & t, int c) {t.channel = c;}
 #endif
```

如果在 tv.cpp 和 use_tv.cpp 中包含 tvfm.h 而不是 tv.h，程序的行为与前一个程序相同，区别在于，只有一个 Remote 方法是 Tv 类的友元，而在原来的版本中，所有的 Remote 方法都是 Tv 类的友元。图 15.1 说明了这种区别。

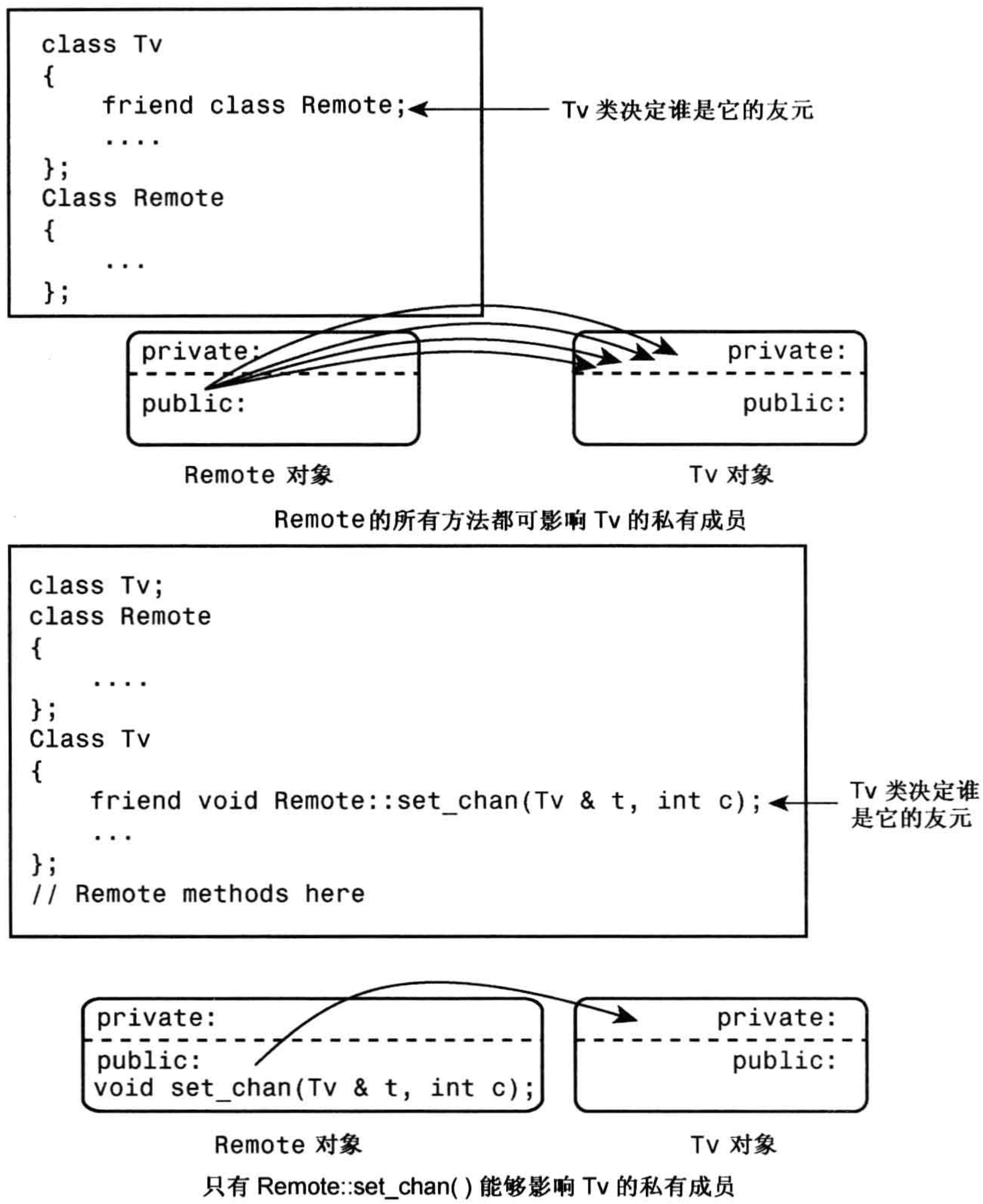

图 15.1　类友元与类成员友元

本书前面介绍过，内联函数的链接性是内部的，这意味着函数定义必须在使用函数的文件中。在这个例子中，内联定义位于头文件中，因此在使用函数的文件中包含头文件可确保将定义放在正确的地方。也可以将定义放在实现文件中，但必须删除关键字 inline，这样函数的链接性将是外部的。

顺便说一句，让整个 Remote 类成为友元并不需要前向声明，因为友元语句本身已经指出 Remote 是一个类：

```
friend class Remote;
```

15.1.3　其他友元关系

除本章前面讨论的，还有其他友元和类的组合形式，下面简要地介绍其中的一些。

假设由于技术进步，出现了交互式遥控器。例如，交互式遥控器让您能够回答电视节目中的问题，如果回答错误，电视将在控制器上产生嗡嗡声。忽略电视使用这种设施安排观众进入节目的可能性，我们只看 C++ 的编程方面。新的方案将受益于相互的友情，一些 Remote 方法能够像前面那样影响 Tv 对象，而一些 Tv 方法也能影响 Remote 对象。这可以通过让类彼此成为对方的友元来实现，即除了 Remote 是 Tv 的友元外，Tv

还是 Remote 的友元。需要记住的一点是，对于使用 Remote 对象的 Tv 方法，其原型可在 Remote 类声明之前声明，但必须在 Remote 类声明之后定义，以便编译器有足够的信息来编译该方法。这种方案与下面类似：

```
class Tv
{
friend class Remote;
public:
    void buzz(Remote & r);
    ...
};
class Remote
{
friend class Tv;
public:
    void Bool volup(Tv & t) { t.volup(); }
    ...
};
inline void Tv::buzz(Remote & r)
{
   ...
}
```

由于 Remote 的声明位于 Tv 声明的后面，所以可以在类声明中定义 Remote::volup( )，但 Tv::buzz( )方法必须在 Tv 声明的外部定义，使其位于 Remote 声明的后面。如果不希望 buzz( )是内联的，则应在一个单独的方法定义文件中定义它。

15.1.4 共同的友元

需要使用友元的另一种情况是，函数需要访问两个类的私有数据。从逻辑上看，这样的函数应是每个类的成员函数，但这是不可能的。它可以是一个类的成员，同时是另一个类的友元，但有时将函数作为两个类的友元更合理。例如，假定有一个 Probe 类和一个 Analyzer 类，前者表示某种可编程的测量设备，后者表示某种可编程的分析设备。这两个类都有内部时钟，且希望它们能够同步，则应该包含下述代码行：

```
class Analyzer;  // forward declaration
class Probe
{
    friend void sync(Analyzer & a, const Probe & p);  // sync a to p
    friend void sync(Probe & p, const Analyzer & a);  // sync p to a
    ...
};
class Analyzer
{
    friend void sync(Analyzer & a, const Probe & p);  // sync a to p
    friend void sync(Probe & p, const Analyzer & a);  // sync p to a
    ...
};

// define the friend functions
inline void sync(Analyzer & a, const Probe & p)
{
    ...
}
inline void sync(Probe & p, const Analyzer & a)
{
    ...
}
```

前向声明使编译器看到 Probe 类声明中的友元声明时，知道 Analyzer 是一种类型。

15.2　嵌套类

在 C++中，可以将类声明放在另一个类中。在另一个类中声明的类被称为嵌套类（nested class），它通过提供新的类型类作用域来避免名称混乱。包含类的成员函数可以创建和使用被嵌套类的对象；而仅当声明位于公有部分，才能在包含类的外面使用嵌套类，而且必须使用作用域解析运算符（然而，旧版本的 C++不允许嵌套类或无法完全实现这种概念）。

对类进行嵌套与包含并不同。包含意味着将类对象作为另一个类的成员，而对类进行嵌套不创建类成员，而是定义了一种类型，该类型仅在包含嵌套类声明的类中有效。

对类进行嵌套通常是为了帮助实现另一个类，并避免名称冲突。Queue 类示例（第 12 章的程序清单 12.8）嵌套了结构定义，从而实现了一种变相的嵌套类：

```
class Queue
{
private:
// class scope definitions
    // Node is a nested structure definition local to this class
    struct Node {Item item; struct Node * next;};
    ...
};
```

由于结构是一种其成员在默认情况下为公有的类，所以 Node 实际上是一个嵌套类，但该定义并没有充分利用类的功能。具体地说，它没有显式构造函数，下面进行补救。

首先，找到 Queue 示例中创建 Node 对象的位置。从类声明（程序清单 11.10）和方法定义（程序清单 12.11）可知，唯一创建了 Node 对象的地方是 enqueue( )方法：

```
bool Queue::enqueue(const Item & item)
{
    if (isfull())
       return false;
    Node * add = new Node;  // create node
// on failure, new throws std::bad_alloc exception
    add->item = item;       // set node pointers
    add->next = NULL;
    ...
}
```

上述代码创建 Node 后，显式地给 Node 成员赋值，这种工作更适合由构造函数来完成。

知道应在什么地方以及如何使用构造函数后，便可以提供一个适当的构造函数定义：

```
class Queue
{
// class scope definitions
    // Node is a nested class definition local to this class
    class Node
    {
    public:
        Item item;
        Node * next;
        Node(const Item & i) : item(i), next(0) { }
    };
    ...
};
```

该构造函数将节点的 item 成员初始化为 i，并将 next 指针设置为 0，这是使用 C++编写空值指针的方法之一（使用 NULL 时，必须包含一个定义 NULL 的头文件；如果您使用的编译器支持 C++11，可使用 nullptr）。由于使用 Queue 类创建的所有节点的 next 的初始值都被设置为空指针，因此这个类只需要该构造函数。

接下来，需要使用构造函数重新编写 enqueue( )：

```
bool Queue::enqueue(const Item & item)
{
    if (isfull())
        return false;
    Node * add = new Node(item);  // create, initialize node
// on failure, new throws std::bad_alloc exception
    ...
}
```

这使得 enqueue( )的代码更短，也更安全，因为它自动进行初始化，无需程序员记住应做什么。

这个例子在类声明中定义了构造函数。假设想在方法文件中定义构造函数，则定义必须指出 Node 类是在 Queue 类中定义的。这是通过使用两次作用域解析运算符来完成的：

```
Queue::Node::Node(const Item & i) : item(i), next(0) { }
```

15.2.1 嵌套类和访问权限

有两种访问权限适合于嵌套类。首先，嵌套类的声明位置决定了嵌套类的作用域，即它决定了程序的哪些部分可以创建这种类的对象。其次，和其他类一样，嵌套类的公有部分、保护部分和私有部分控制了对类成员的访问。在哪些地方可以使用嵌套类以及如何使用嵌套类，取决于作用域和访问控制。下面将更详细地进行介绍。

1. 作用域

如果嵌套类是在另一个类的私有部分声明的，则只有后者知道它。在前一个例子中，被嵌套在 Queue 声明中的 Node 类就属于这种情况（看起来 Node 是在私有部分之前定义的，但别忘了，类的默认访问权限是私有的），因此，Queue 成员可以使用 Node 对象和指向 Node 对象的指针，但是程序的其他部分甚至不知道存在 Node 类。对于从 Queue 派生而来的类，Node 也是不可见的，因为派生类不能直接访问基类的私有部分。

如果嵌套类是在另一个类的保护部分声明的，则它对于后者来说是可见的，但是对于外部世界则是不可见的。然而，在这种情况中，派生类将知道嵌套类，并可以直接创建这种类型的对象。

如果嵌套类是在另一个类的公有部分声明的，则允许后者、后者的派生类以及外部世界使用它，因为它是公有的。然而，由于嵌套类的作用域为包含它的类，因此在外部世界使用它时，必须使用类限定符。例如，假设有下面的声明：

```
class Team
{
public:
      class Coach { ... };
      ...
};
```

现在假定有一个失业的教练，他不属于任何球队。要在 Team 类的外面创建 Coach 对象，可以这样做：

```
Team::Coach forhire;  // create a Coach object outside the Team class
```

嵌套结构和枚举的作用域与此相同。其实，很多程序员都使用公有枚举来提供可供客户程序员使用的类常数。例如，很多类实现都被定义为支持 iostream 使用这种技术来提供不同的格式选项，前面已经介绍过这方面的内容，第 17 章将更加全面地进行介绍。表 15.1 总结了嵌套类、结构和枚举的作用域特征。

表 15.1　嵌套类、结构和枚举的作用域特征

声明位置	包含它的类是否可以使用它	从包含它的类派生而来的类是否可以使用它	在外部是否可以使用
私有部分	是	否	否
保护部分	是	是	否
公有部分	是	是	是，通过类限定符来使用

2. 访问控制

类可见后，起决定作用的将是访问控制。对嵌套类访问权的控制规则与对常规类相同。在 Queue 类声明中声明 Node 类并没有赋予 Queue 类任何对 Node 类的访问特权，也没有赋予 Node 类任何对 Queue 类的访问特权。因此，Queue 类对象只能显示地访问 Node 对象的公有成员。由于这个原因，在 Queue 示例中，Node 类的所有成员都被声明为公有的。这样有悖于应将数据成员声明为私有的这一惯例，但 Node 类是 Queue 类内部实现的一项特性，对外部世界是不可见的。这是因为 Node 类是在 Queue 类的私有部分声明的。所以，虽然 Queue 的方法可直接访问 Node 的成员，但使用 Queue 类的客户不能这样做。

总之，类声明的位置决定了类的作用域或可见性。类可见后，访问控制规则（公有、保护、私有、友元）将决定程序对嵌套类成员的访问权限。

15.2.2　模板中的嵌套

您知道，模板很适合用于实现诸如 Queue 等容器类。您可能会问，将 Queue 类定义转换为模板时，是否会由于它包含嵌套类而带来问题？答案是不会。程序清单 15.5 演示了如何进行这种转换。和类模板一样，该头文件也包含类模板和方法函数模板。

程序清单 15.5　queuetp.h

```
// queuetp.h -- queue template with a nested class
#ifndef QUEUETP_H_
#define QUEUETP_H_

template <class Item>
class QueueTP
{
private:
    enum {Q_SIZE = 10};
    // Node is a nested class definition
    class Node
    {
    public:
        Item item;
        Node * next;
        Node(const Item & i):item(i), next(0){ }
    };
    Node * front;       // pointer to front of Queue
    Node * rear;        // pointer to rear of Queue
    int items;          // current number of items in Queue
    const int qsize;    // maximum number of items in Queue
    QueueTP(const QueueTP & q) : qsize(0) {}
    QueueTP & operator=(const QueueTP & q) { return *this; }
public:
    QueueTP(int qs = Q_SIZE);
    ~QueueTP();
    bool isempty() const
    {
```

```
        return items == 0;
    }
    bool isfull() const
    {
        return items == qsize;
    }
    int queuecount() const
    {
        return items;
    }
    bool enqueue(const Item &item); // add item to end
    bool dequeue(Item &item);       // remove item from front
};
// QueueTP methods
template <class Item>
QueueTP<Item>::QueueTP(int qs) : qsize(qs)
{
    front = rear = 0;
    items = 0;
}

template <class Item>
QueueTP<Item>::~QueueTP()
{
    Node * temp;
    while (front != 0)      // while queue is not yet empty
    {
        temp = front;       // save address of front item
        front = front->next;// reset pointer to next item
        delete temp;        // delete former front
    }
}

// Add item to queue
template <class Item>
bool QueueTP<Item>::enqueue(const Item & item)
{
    if (isfull())
        return false;
    Node * add = new Node(item);    // create node
// on failure, new throws std::bad_alloc exception
    items++;
    if (front == 0)         // if queue is empty,
        front = add;        // place item at front
    else
        rear->next = add;   // else place at rear
    rear = add;             // have rear point to new node
    return true;
}

// Place front item into item variable and remove from queue
template <class Item>
bool QueueTP<Item>::dequeue(Item & item)
{
    if (front == 0)
```

```
    return false;
item = front->item;      // set item to first item in queue
items--;
Node * temp = front;     // save location of first item
front = front->next;     // reset front to next item
delete temp;             // delete former first item
if (items == 0)
    rear = 0;
return true;
}

#endif
```

程序清单 15.5 中模板有趣的一点是，Node 是利用通用类型 Item 来定义的。所以，下面的声明将导致 Node 被定义成用于存储 double 值：

```
QueueTp<double> dq;
```

而下面的声明将导致 Node 被定义成用于存储 char 值：

```
QueueTp<char> cq;
```

这两个 Node 类将在两个独立的 QueueTP 类中定义，因此不会发生名称冲突。即一个节点的类型为 QueueTP<double>::Node，另一个节点的类型为 QueueTP<char>::Node。

程序清单 15.6 是一个小程序，可用于测试这个新的类。

程序清单 15.6　nested.cpp

```
// nested.cpp -- using a queue that has a nested class
#include <iostream>

#include <string>
#include "queuetp.h"

int main()
{
    using std::string;
    using std::cin;
    using std::cout;

    QueueTP<string> cs(5);
    string temp;

    while(!cs.isfull())
    {
        cout << "Please enter your name. You will be "
                "served in the order of arrival.\n"
                "name: ";
        getline(cin, temp);
        cs.enqueue(temp);
    }
    cout << "The queue is full. Processing begins!\n";

    while (!cs.isempty())
    {
        cs.dequeue(temp);
        cout << "Now processing " << temp << "...\n";
```

```
    }
    return 0;
}
```

程序清单 15.5 和程序清单 15.6 组成的程序的运行情况如下：

```
Please enter your name. You will be served in the order of arrival.
name: Kinsey Millhone
Please enter your name. You will be served in the order of arrival.
name: Adam Dalgliesh
Please enter your name. You will be served in the order of arrival.
name: Andrew Dalziel
Please enter your name. You will be served in the order of arrival.
name: Kay Scarpetta
Please enter your name. You will be served in the order of arrival.
name: Richard Jury
The queue is full. Processing begins!
Now processing Kinsey Millhone...
Now processing Adam Dalgliesh...
Now processing Andrew Dalziel...
Now processing Kay Scarpetta...
Now processing Richard Jury...
```

15.3 异常

程序有时会遇到运行阶段错误，导致程序无法正常地运行下去。例如，程序可能试图打开一个不可用的文件，请求过多的内存，或者遭遇不能容忍的值。通常，程序员都会试图预防这种意外情况。C++异常为处理这种情况提供了一种功能强大而灵活的工具。异常是相对较新的 C++功能，有些老式编译器可能没有实现。另外，有些编译器默认关闭这种特性，您可能需要使用编译器选项来启用它。

讨论异常之前，先来看看程序员可使用的一些基本方法。作为试验，以一个计算两个数的调和平均数的函数为例。两个数的调和平均数的定义是：这两个数字倒数的平均值的倒数，因此表达式为：

```
2.0 × x × y / (x + y)
```

如果 y 是 x 的负值，则上述公式将导致被零除——一种不允许的运算。对于被零除的情况，很多新式编译器通过生成一个表示无穷大的特殊浮点值来处理，cout 将这种值显示为 Inf、inf、INF 或类似的东西；而其他的编译器可能生成在发生被零除时崩溃的程序。最好编写在所有系统上都以相同的受控方式运行的代码。

15.3.1 调用 abort( )

对于这种问题，处理方式之一是，如果其中一个参数是另一个参数的负值，则调用 abort( )函数。Abort( )函数的原型位于头文件 cstdlib（或 stdlib.h）中，其典型实现是向标准错误流（即 cerr 使用的错误流）发送消息 abnormal program termination（程序异常终止），然后终止程序。它还返回一个随实现而异的值，告诉操作系统（如果程序是由另一个程序调用的，则告诉父进程），处理失败。abort( )是否刷新文件缓冲区（用于存储读写到文件中的数据的内存区域）取决于实现。如果愿意，也可以使用 exit( )，该函数刷新文件缓冲区，但不显示消息。程序清单 15.7 是一个使用 abort( )的小程序。

程序清单 15.7　error1.cpp

```
//error1.cpp -- using the abort() function
#include <iostream>
#include <cstdlib>
double hmean(double a, double b);
```

```
int main()
{
    double x, y, z;

    std::cout << "Enter two numbers: ";
    while (std::cin >> x >> y)
    {
        z = hmean(x,y);
        std::cout << "Harmonic mean of " << x << " and " << y
            << " is " << z << std::endl;
        std::cout << "Enter next set of numbers <q to quit>: ";
    }
    std::cout << "Bye!\n";
    return 0;
}

double hmean(double a, double b)
{
    if (a == -b)
    {
        std::cout << "untenable arguments to hmean()\n";
        std::abort();
    }
    return 2.0 * a * b / (a + b);
}
```

程序清单 15.7 中程序的运行情况如下：

```
Enter two numbers: 3 6
Harmonic mean of 3 and 6 is 4
Enter next set of numbers <q to quit>: 10 -10
untenable arguments to hmean()
abnormal program termination
```

注意，在 hmean( )中调用 abort( )函数将直接终止程序，而不是先返回到 main( )。一般而言，显示的程序异常中断消息随编译器而异，下面是另一种编译器显示的消息：

```
This application has requested the Runtime to terminate it
in an unusual way. Please contact the application's support
team for more information.
```

为了避免异常终止，程序应在调用 hmean( )函数之前检查 x 和 y 的值。然而，依靠程序员来执行这种检查是不安全的。

15.3.2　返回错误码

一种比异常终止更灵活的方法是，使用函数的返回值来指出问题。例如，ostream 类的 get（void）成员通常返回下一个输入字符的 ASCII 码，但到达文件尾时，将返回特殊值 EOF。对 hmean( )来说，这种方法不管用。任何数值都是有效的返回值，因此不存在可用于指出问题的特殊值。在这种情况下，可使用指针参数或引用参数来将值返回给调用程序，并使用函数的返回值来指出成功还是失败。istream 族重载>>运算符使用了这种技术的变体。通过告知调用程序是成功了还是失败了，使得程序可以采取除异常终止程序之外的其他措施。程序清单 15.8 是一个采用这种方式的示例，它将 hmean( )的返回值重新定义为 bool，让返回值指出成功了还是失败了，另外还给该函数增加了第三个参数，用于提供答案。

程序清单 15.8 error2.cpp

```
//error2.cpp -- returning an error code
#include <iostream>
#include <cfloat>  // (or float.h) for DBL_MAX

bool hmean(double a, double b, double * ans);

int main()
{
    double x, y, z;

    std::cout << "Enter two numbers: ";
    while (std::cin >> x >> y)
    {
        if (hmean(x,y,&z))
            std::cout << "Harmonic mean of " << x << " and " << y
                << " is " << z << std::endl;
        else
            std::cout << "One value should not be the negative "
                << "of the other - try again.\n";
        std::cout << "Enter next set of numbers <q to quit>: ";
    }
    std::cout << "Bye!\n";
    return 0;
}

bool hmean(double a, double b, double * ans)
{
    if (a == -b)
    {
        *ans = DBL_MAX;
        return false;
    }
    else
    {
        *ans = 2.0 * a * b / (a + b);
        return true;
    }
}
```

程序清单 15.8 中程序的运行情况如下：

```
Enter two numbers: 3 6
Harmonic mean of 3 and 6 is 4
Enter next set of numbers <q to quit>: 10 -10
One value should not be the negative of the other - try again.
Enter next set of numbers <q to quit>: 1 19
Harmonic mean of 1 and 19 is 1.9
Enter next set of numbers <q to quit>: q
Bye!
```

程序说明

在程序清单 15.8 中，程序设计避免了错误输入导致的恶果，让用户能够继续输入。当然，设计确实依靠用户检查函数的返回值，这项工作是程序员所不经常做的。例如，为使程序短小精悍，本书的程序清单

都没有检查 cout 是否成功地处理了输出。

第三参数可以是指针或引用。对内置类型的参数，很多程序员都倾向于使用指针，因为这样可以明显看出是哪个参数用于提供答案。

另一种在某个地方存储返回条件的方法是使用一个全局变量。可能问题的函数可以在出现问题时将该全局变量设置为特定的值，而调用程序可以检查该变量。传统的 C 语言数学库使用的就是这种方法，它使用的全局变量名为 errno。当然，必须确保其他函数没有将该全局变量用于其他目的。

15.3.3　异常机制

下面介绍如何使用异常机制来处理错误。C++异常是对程序运行过程中发生的异常情况（例如被 0 除）的一种响应。异常提供了将控制权从程序的一个部分传递到另一部分的途径。对异常的处理有 3 个组成部分：

- 引发异常；
- 使用处理程序捕获异常；
- 使用 try 块。

程序在出现问题时将引发异常。例如，可以修改程序清单 15.7 中的 hmean( )，使之引发异常，而不是调用 abort( )函数。throw 语句实际上是跳转，即命令程序跳到另一条语句。throw 关键字表示引发异常，紧随其后的值（例如字符串或对象）指出了异常的特征。

程序使用异常处理程序（exception handler）来捕获异常，异常处理程序位于要处理问题的程序中。catch 关键字表示捕获异常。处理程序以关键字 catch 开头，随后是位于括号中的类型声明，它指出了异常处理程序要响应的异常类型；然后是一个用花括号括起的代码块，指出要采取的措施。catch 关键字和异常类型用作标签，指出当异常被引发时，程序应跳到这个位置执行。异常处理程序也被称为 catch 块。

try 块标识其中特定的异常可能被激活的代码块，它后面跟一个或多个 catch 块。try 块是由关键字 try 指示的，关键字 try 的后面是一个由花括号括起的代码块，表明需要注意这些代码引发的异常。

要了解这 3 个元素是如何协同工作的，最简单的方法是看一个简短的例子，如程序清单 15.9 所示。

程序清单 15.9　error3.cpp

```
// error3.cpp -- using an exception
#include <iostream>
double hmean(double a, double b);

int main()
{
    double x, y, z;

    std::cout << "Enter two numbers: ";
    while (std::cin >> x >> y)
    {
        try {                    // start of try block
            z = hmean(x,y);
        }                        // end of try block
        catch (const char * s)   // start of exception handler
        {
            std::cout << s << std::endl;
            std::cout << "Enter a new pair of numbers: ";
            continue;
        }                        // end of handler
        std::cout << "Harmonic mean of " << x << " and " << y
            << " is " << z << std::endl;
        std::cout << "Enter next set of numbers <q to quit>: ";
    }
    std::cout << "Bye!\n";
```

```
    return 0;
}

double hmean(double a, double b)
{
    if (a == -b)
        throw "bad hmean() arguments: a = -b not allowed";
    return 2.0 * a * b / (a + b);
}
```

程序清单 15.9 中程序的运行情况如下：

```
Enter two numbers: 3 6
Harmonic mean of 3 and 6 is 4
Enter next set of numbers <q to quit>: 10 -10
bad hmean() arguments: a = -b not allowed
Enter a new pair of numbers: 1 19
Harmonic mean of 1 and 19 is 1.9
Enter next set of numbers <q to quit>: q
Bye!
```

程序说明

在程序清单 15.9 中，try 块与下面类似：

```
try {                    // start of try block
    z = hmean(x,y);
}                        // end of try block
```

如果其中的某条语句导致异常被引发，则后面的 catch 块将对异常进行处理。如果程序在 try 块的外面调用 hmean( )，将无法处理异常。

引发异常的代码与下面类似：

```
if (a == -b)
    throw "bad hmean() arguments: a = -b not allowed";
```

其中被引发的异常是字符串“bad hmean( )arguments: a = -b not allowed”。异常类型可以是字符串（就像这个例子中那样）或其他 C++类型；通常为类类型，本章后面的示例将说明这一点。

执行 throw 语句类似于执行返回语句，因为它也将终止函数的执行；但 throw 不是将控制权返回给调用程序，而是导致程序沿函数调用序列后退，直到找到包含 try 块的函数。在程序清单 15.9 中，该函数是调用函数。稍后将有一个沿函数调用序列后退多步的例子。另外，在这个例子中，throw 将程序控制权返回给 main( )。程序将在 main( )中寻找与引发的异常类型匹配的异常处理程序（位于 try 块的后面）。

处理程序（或 catch 块）与下面类似：

```
catch (char * s)  // start of exception handler
{
    std::cout << s << std::endl;
    sdt::cout << "Enter a new pair of numbers: ";
    continue;
}                        // end of handler
```

catch 块点类似于函数定义，但并不是函数定义。关键字 catch 表明这是一个处理程序，而 char*s 则表明该处理程序与字符串异常匹配。s 与函数参数定义极其类似，因为匹配的引发将被赋给 s。另外，当异常与该处理程序匹配时，程序将执行括号中的代码。

执行完 try 块中的语句后，如果没有引发任何异常，则程序跳过 try 块后面的 catch 块，直接执行处理程序后面的第一条语句。因此处理值 3 和 6 时，程序清单 15.9 中程序执行报告结果的输出语句。

接下来看将 10 和−10 传递给 hmean( )函数后发生的情况。If 语句导致 hmean( )引发异常。这将终止

hmean( )的执行。程序向后搜索时发现，hmean( )函数是从 main( )中的 try 块中调用的，因此程序查找与异常类型匹配的 catch 块。程序中唯一的一个 catch 块的参数为 char*，因此它与引发异常匹配。程序将字符串“bad hmean( )arguments: a = -b not allowed”赋给变量 s，然后执行处理程序中的代码。处理程序首先打印 s——捕获的异常，然后打印要求用户输入新数据的指示，最后执行 continue 语句，命令程序跳过 while 循环的剩余部分，跳到起始位置。continue 使程序跳到循环的起始处，这表明处理程序语句是循环的一部分，而 catch 行是指引程序流程的标签（参见图 15.2）。

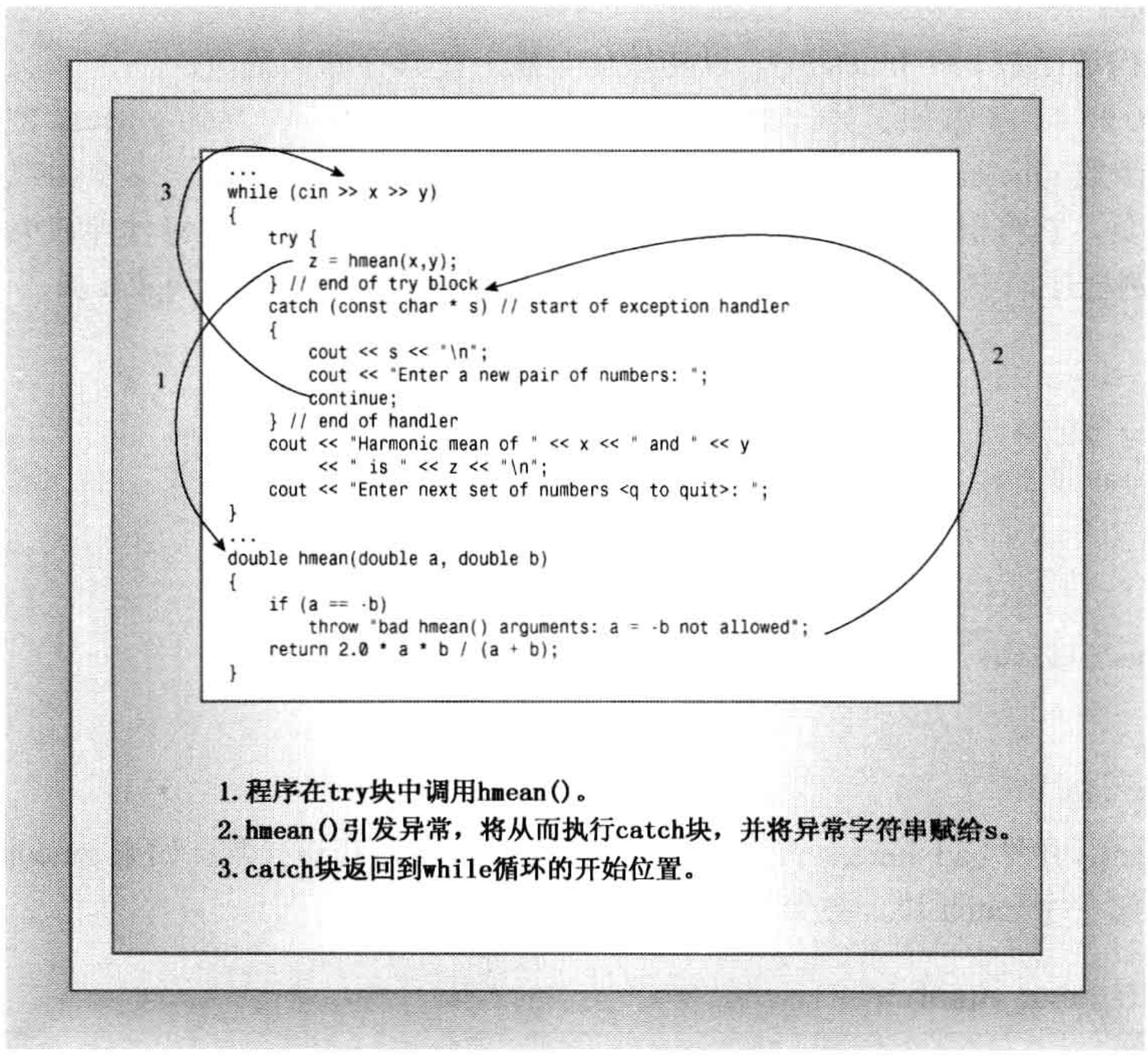

图 15.2　出现异常时的程序流程

您可能会问，如果函数引发了异常，而没有 try 块或没有匹配的处理程序时，将会发生什么情况。在默认情况下下，程序最终将调用 abort( )函数，但可以修改这种行为。稍后将讨论这个问题。

15.3.4　将对象用作异常类型

通常，引发异常的函数将传递一个对象。这样做的重要优点之一是，可以使用不同的异常类型来区分不同的函数在不同情况下引发的异常。另外，对象可以携带信息，程序员可以根据这些信息来确定引发异常的原因。同时，catch 块可以根据这些信息来决定采取什么样的措施。例如，下面是针对函数 hmean( )引发的异常而提供的一种设计：

```
class bad_hmean
{
private:
    double v1;
    double v2;
public:
    bad_hmean(int a = 0, int b = 0) : v1(a), v2(b){}
    void mesg();
};

inline void bad_hmean::mesg()
{
```

```
    std::cout << "hmean(" << v1 << ", " << v2 <<"): "
              << "invalid arguments: a = -b\n";
}
```

可以将一个 bad_hmean 对象初始化为传递给函数 hmean( )的值，而方法 mesg( )可用于报告问题（包括传递给函数 hmena( )的值）。函数 hmean( )可以使用下面这样的代码：

```
if (a == -b)
    throw bad_hmean(a,b);
```

上述代码调用构造函数 bad_hmean( )，以初始化对象，使其存储参数值。

程序清单 15.10 和 15.11 添加了另一个异常类 bad_gmean 以及另一个名为 gmean( )的函数，该函数引发 bad_gmean 异常。函数 gmean( )计算两个数的几何平均值，即乘积的平方根。这个函数要求两个参数都不为负，如果参数为负，它将引发异常。程序清单 15.10 是一个头文件，其中包含异常类的定义；而程序清单 15.11 是一个示例程序，它使用了该头文件。注意，try 块的后面跟着两个 catch 块：

```
try {                   // start of try block
    ...
}// end of try block
catch (bad_hmean & bg)     // start of catch block
{
   ...
}
catch (bad_gmean & hg)
{
   ...
} // end of catch block
```

如果函数 hmean( )引发 bad_hmean 异常，第一个 catch 块将捕获该异常；如果 gmean( )引发 bad_gmean 异常，异常将逃过第一个 catch 块，被第二个 catch 块捕获。

程序清单 15.10　exc_mean.h

```
// exc_mean.h  -- exception classes for hmean(), gmean()
#include <iostream>

class bad_hmean
{
private:
    double v1;
    double v2;
public:
    bad_hmean(double a = 0, double b = 0) : v1(a), v2(b){}
    void mesg();
};

inline void bad_hmean::mesg()
{
    std::cout << "hmean(" << v1 << ", " << v2 <<"): "
              << "invalid arguments: a = -b\n";
}
class bad_gmean
{
public:
    double v1;
    double v2;
    bad_gmean(double a = 0, double b = 0) : v1(a), v2(b){}
```

```
    const char * mesg();
};

inline const char * bad_gmean::mesg()
{
    return "gmean() arguments should be >= 0\n";
}
```

程序清单 15.11　error4.cpp

```
//error4.cpp - using exception classes
#include <iostream>
#include <cmath> // or math.h, unix users may need -lm flag
#include "exc_mean.h"
// function prototypes
double hmean(double a, double b);
double gmean(double a, double b);
int main()
{
    using std::cout;
    using std::cin;
    using std::endl;

    double x, y, z;

    cout << "Enter two numbers: ";
    while (cin >> x >> y)
    {
        try {                    // start of try block
            z = hmean(x,y);
            cout << "Harmonic mean of " << x << " and " << y
                << " is " << z << endl;
            cout << "Geometric mean of " << x << " and " << y
                << " is " << gmean(x,y) << endl;
            cout << "Enter next set of numbers <q to quit>: ";
        }// end of try block
        catch (bad_hmean & bg)    // start of catch block
        {
            bg.mesg();
            cout << "Try again.\n";
            continue;
        }
        catch (bad_gmean & hg)
        {
            cout << hg.mesg();
            cout << "Values used: " << hg.v1 << ", "
                 << hg.v2 << endl;
            cout << "Sorry, you don't get to play any more.\n";
            break;
        } // end of catch block
    }
    cout << "Bye!\n";
    return 0;
}
```

```
double hmean(double a, double b)
{
    if (a == -b)
        throw bad_hmean(a,b);
    return 2.0 * a * b / (a + b);
}

double gmean(double a, double b)
{
    if (a < 0 || b < 0)
        throw bad_gmean(a,b);
    return std::sqrt(a * b);
}
```

下面是程序清单 15.10 和 15.11 组成的程序的运行情况，错误的 gmean( )函数输入导致程序终止：

```
Enter two numbers: 4 12
Harmonic mean of 4 and 12 is 6
Geometric mean of 4 and 12 is 6.9282
Enter next set of numbers <q to quit>: 5 -5
hmean(5, -5): invalid arguments: a = -b
Try again.
5 -2
Harmonic mean of 5 and -2 is -6.66667
gmean() arguments should be >= 0
Values used: 5, -2
Sorry, you don't get to play any more.
Bye!
```

首先，bad_hmean 异常处理程序使用了一条 continue 语句，而 bad_gmean 异常处理程序使用了一条 break 语句。因此，如果用户给函数 hmean( )提供的参数不正确，将导致程序跳过循环中余下的代码，进入下一次循环；而用户给函数 gmean( )提供的参数不正确时将结束循环。这演示了程序如何确定引发的异常（根据异常类型）并据此采取相应的措施。

其次，异常类 bad_gmean 和 bad_hmean 使用的技术不同，具体地说，bad_gmean 使用的是公有数据和一个公有方法，该方法返回一个 C-风格字符串。

15.3.5 异常规范和 C++11

有时候，一种理念看似有前途，但实际的使用效果并不好。一个这样的例子是异常规范（exception specification），这是 C++98 新增的一项功能，但 C++11 却将其摒弃了。这意味着 C++11 仍然处于标准之中，但以后可能会从标准中剔除，因此不建议您使用它。

然而，忽视异常规范前，您至少应该知道它是什么样的，如下所示：

```
double harm(double a) throw(bad_thing);  // may throw bad_thing exception
double marm(double) throw();             // doesn't throw an exception
```

其中的 throw( )部分就是异常规范，它可能出现在函数原型和函数定义中，可包含类型列表，也可不包含。

异常规范的作用之一是，告诉用户可能需要使用 try 块。然而，这项工作也可使用注释轻松地完成。异常规范的另一个作用是，让编译器添加执行运行阶段检查的代码，检查是否违反了异常规范。这很难检查。例如，marm( )可能不会引发异常，但它可能调用一个函数，而这个函数调用的另一个函数引发了异常。另外，您给函数编写代码时它不会引发异常，但库更新后它却会引发异常。总之，编程社区（尤其是尽力编写安全代码的开发人员）达成的一致意见是，最好不要使用这项功能。而 C++11 也建议您忽略异常规范。

然而，C++11 确实支持一种特殊的异常规范：您可使用新增的关键字 noexcept 指出函数不会引发异常：

```
double marm() noexcept;  // marm() doesn't throw an exception
```

有关这种异常规范是否必要和有用存在一些争议，有些人认为最好不要使用它（至少在大多数情况下如此）；而有些人认为引入这个新关键字很有必要，理由是知道函数不会引发异常有助于编译器优化代码。通过使用这个关键字，编写函数的程序员相当于做出了承诺。

还有运算符 noexcept( )，它判断其操作数是否会引发异常，详情请参阅附录 E。

15.3.6　栈解退

假设 try 块没有直接调用引发异常的函数，而是调用了对引发异常的函数进行调用的函数，则程序流程将从引发异常的函数跳到包含 try 块和处理程序的函数。这涉及到栈解退（unwinding the stack），下面进行介绍。

首先来看一看 C++通常是如何处理函数调用和返回的。C++通常通过将信息放在栈（参见第 9 章）中来处理函数调用。具体地说，程序将调用函数的指令的地址（返回地址）放到栈中。当被调用的函数执行完毕后，程序将使用该地址来确定从哪里开始继续执行。另外，函数调用将函数参数放到栈中。在栈中，这些函数参数被视为自动变量。如果被调用的函数创建了新的自动变量，则这些变量也将被添加到栈中。如果被调用的函数调用了另一个函数，则后者的信息将被添加到栈中，依此类推。当函数结束时，程序流程将跳到该函数被调用时存储的地址处，同时栈顶的元素被释放。因此，函数通常都返回到调用它的函数，依此类推，同时每个函数都在结束时释放其自动变量。如果自动变量是类对象，则类的析构函数（如果有的话）将被调用。

现在假设函数由于出现异常（而不是由于返回）而终止，则程序也将释放栈中的内存，但不会在释放栈的第一个返回地址后停止，而是继续释放栈，直到找到一个位于 try 块（参见图 15.3）中的返回地址。随后，控制权将转到块尾的异常处理程序，而不是函数调用后面的第一条语句。这个过程被称为栈解退。引发机制的一个非常重要的特性是，和函数返回一样，对于栈中的自动类对象，类的析构函数将被调用。然而，函数返回仅仅处理该函数放在栈中的对象，而 throw 语句则处理 try 块和 throw 之间整个函数调用序列放在栈中的对象。如果没有栈解退这种特性，则引发异常后，对于中间函数调用放在栈中的自动类对象，其析构函数将不会被调用。

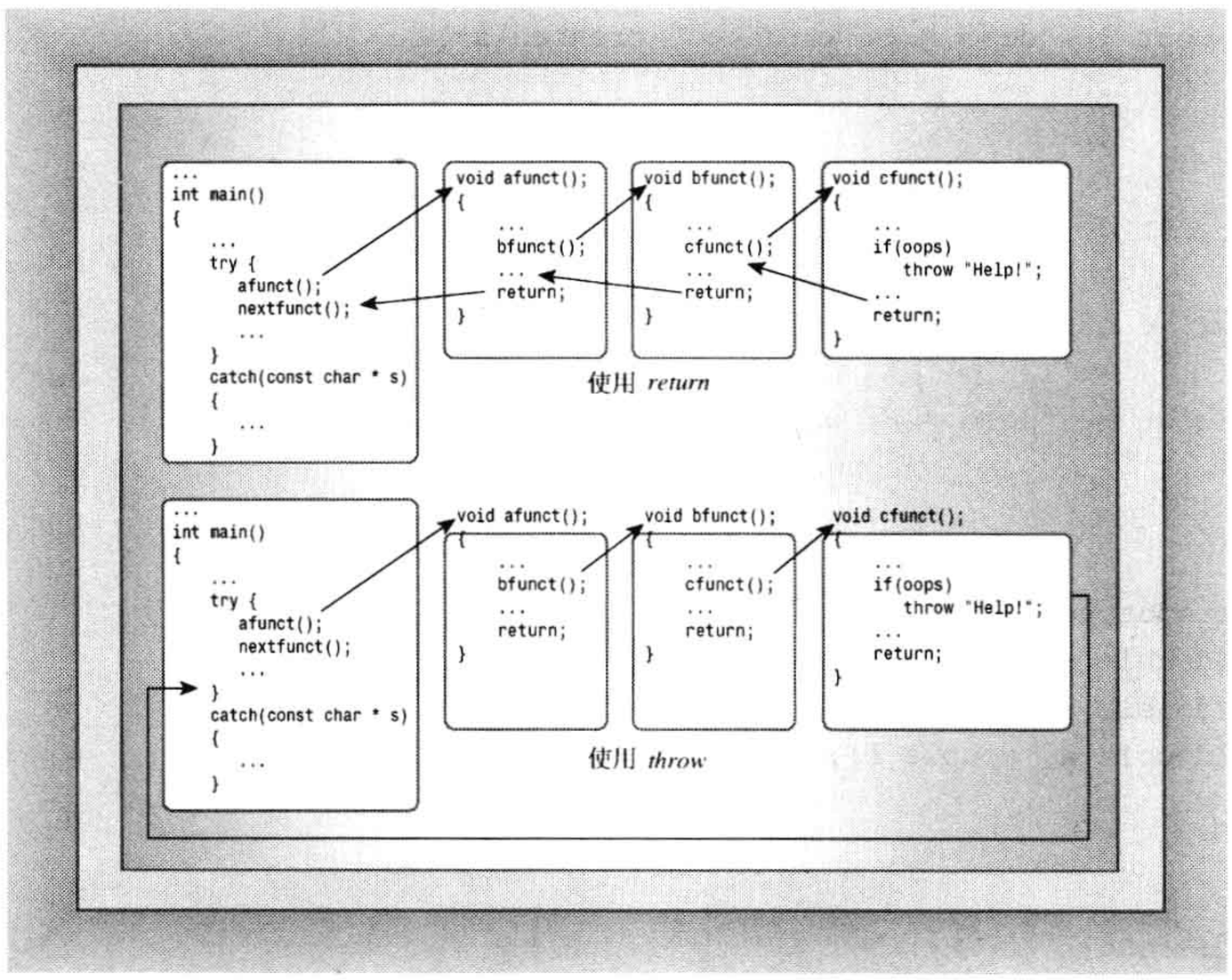

图 15.3　throw 与 return

程序清单 15.12 是一个栈解退的示例。其中，main( )调用了 means( )，而 means( )又调用了 hmean( )和 gmean( )。函数 means( )计算算术平均数、调和平均数和几何平均数。main( )和 means( )都创建 demo 类型的对象（demo 是一个喋喋不休的类，指出什么时候构造函数和析构函数被调用），以便您知道发生异常时这些对象将被如何处理。函数 main( )中的 try 块能够捕获 bad_hmean 和 badgmean 异常，而函数 means( )中的 try 块只能捕获 bad_hmean 异常。catch 块的代码如下：

```
catch (bad_hmean & bg) // start of catch block
{
    bg.mesg();
    std::cout << "Caught in means()\n";
    throw;              // rethrows the exception
}
```

上述代码显示消息后，重新引发异常，这将向上把异常发送给 main( )函数。一般而言，重新引发的异常将由下一个捕获这种异常的 try-catch 块组合进行处理，如果没有找到这样的处理程序，默认情况下程序将异常终止。程序清单 15.12 使用的头文件与程序清单 15.11 使用的相同（程序清单 15.10 所示的 exc_mean.h）。

程序清单 15.12　error5.cpp

```
//error5.cpp -- unwinding the stack
#include <iostream>
#include <cmath> // or math.h, unix users may need -lm flag
#include <string>
#include "exc_mean.h"

class demo
{
private:
    std::string word;
public:
    demo (const std::string & str)
    {

        word = str;
        std::cout << "demo " << word << " created\n";
    }
    ~demo()
    {
        std::cout << "demo " << word << " destroyed\n";
    }
    void show() const
    {
        std::cout << "demo " << word << " lives!\n";
    }
};

// function prototypes
double hmean(double a, double b);
double gmean(double a, double b);
double means(double a, double b);

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

    double x, y, z;
    {
        demo d1("found in block in main()");
        cout << "Enter two numbers: ";
        while (cin >> x >> y)
        {
```

```
        try {                    // start of try block
            z = means(x,y);
            cout << "The mean mean of " << x << " and " << y
                    << " is " << z << endl;
            cout << "Enter next pair: ";
        } // end of try block
        catch (bad_hmean & bg)    // start of catch block
        {
            bg.mesg();
            cout << "Try again.\n";
            continue;
        }
        catch (bad_gmean & hg)
        {
            cout << hg.mesg();
            cout << "Values used: " << hg.v1 << ", "
                    << hg.v2 << endl;
            cout << "Sorry, you don't get to play any more.\n";
            break;
        } // end of catch block
    }
    d1.show();
}
cout << "Bye!\n";
cin.get();
cin.get();
return 0;
}

double hmean(double a, double b)
{
    if (a == -b)
        throw bad_hmean(a,b);
    return 2.0 * a * b / (a + b);
}

double gmean(double a, double b)
{
    if (a < 0 || b < 0)
        throw bad_gmean(a,b);
    return std::sqrt(a * b);
}

double means(double a, double b)
{
    double am, hm, gm;
    demo d2("found in means()");
    am = (a + b) / 2.0;    // arithmetic mean
    try
    {
        hm = hmean(a,b);
        gm = gmean(a,b);
    }
    catch (bad_hmean & bg) // start of catch block
    {
        bg.mesg();
```

```
        std::cout << "Caught in means()\n";
        throw;              // rethrows the exception
    }
    d2.show();
    return (am + hm + gm) / 3.0;
}
```

下面是程序清单 15.10 和程序清单 15.12 组成的程序的运行情况：

```
demo found in block in main() created
Enter two numbers: 6 12
demo found in means() created
demo found in means() lives!
demo found in means() destroyed
The mean mean of 6 and 12 is 8.49509
6 -6
demo found in means() created
hmean(6, -6): invalid arguments: a = -b
Caught in means()
demo found in means() destroyed
hmean(6, -6): invalid arguments: a = -b
Try again.
6 -8
demo found in means() created
demo found in means() destroyed
gmean() arguments should be >= 0
Values used: 6, -8
Sorry, you don't get to play any more.
demo found in block in main() lives!
demo found in block in main() destroyed
Bye!
```

程序说明

来看看该程序的运行过程。首先，正如 demo 类的构造函数指出的，在 main( )函数中创建了一个 demo 对象。接下来，调用了函数 means( )，它创建了另一个 demo 对象。函数 means( )使用 6 和 2 来调用函数 hmean( )和 gmean( )，它们将结果返回给 means( )，后者计算一个结果并将其返回。返回结果前，means( )调用了 d2.show( )；返回结果后，函数 means( )执行完毕，因此自动为 d2 调用析构函数：

```
demo found in means() lives!
demo found in means() destroyed
```

接下来的输入循环将值 6 和–6 发送给函数 means( )，然后 means( )创建一个新的 demo 对象，并将值传递给 hmean( )。函数 hmean( )引发 bad_hmean 异常，该异常被 means( )中的 catch 块捕获，下面的输出指出了这一点：

```
hmean(6, -6): invalid arguments: a = -b
Caught in means()
```

该 catch 块中的 throw 语句导致函数 means( )终止执行，并将异常传递给 main( )函数。语句 d2.show( )没有被执行表明 means( )函数被提前终止。但需要指出的是，还是为 d2 调用了析构函数：

```
demo found in means() destroyed
```

这演示了异常极其重要的一点：程序进行栈解退以回到能够捕获异常的地方时，将释放栈中的自动存储型变量。如果变量是类对象，将为该对象调用析构函数。

与此同时，重新引发的异常被传递给 main( )，在该函数中，合适的 catch 块将捕获它并对其进行处理：

```
hmean(6, -6): invalid arguments: a = -b
Try again.
```

接下来开始了第三次输入循环：6 和−8 被发送给函数 means( )。同样，means( )创建一个新的 demo 对象，然后将 6 和−8 传递给 hmean( )，后者在处理它们时没有出现问题。然而，means( )将 6 和−8 传递给 gmean( )，后者引发了 bad_gmean 异常。由于 means( )不能捕获 bad_gmean 异常，因此异常被传递给 main( )，同时不再执行 means( )中的其他代码。同样，当程序进行栈解退时，将释放局部的动态变量，因此为 d2 调用了析构函数：

```
demo found in means() destroyed
```

最后，main( )中的 bad_gmean 异常处理程序捕获了该异常，循环结束：

```
gmean() arguments should be >= 0
Values used: 6, -8
Sorry, you don't get to play any more.
```

然后程序正常终止：显示一些消息并自动为 d1 调用析构函数。如果 catch 块使用的是 exit(EXIT_FAILURE)而不是 break，则程序将立刻终止，用户将看不到下述消息：

```
demo found in main() lives!
Bye!
```

但仍能够看到如下消息：

```
demo found in main() destroyed
```

同样，异常机制将负责释放栈中的自动变量。

15.3.7　其他异常特性

虽然 throw-catch 机制类似于函数参数和函数返回机制，但还是有些不同之处。其中之一是函数 fun( )中的返回语句将控制权返回到调用 fun( )的函数，但 throw 语句将控制权向上返回到第一个这样的函数：包含能够捕获相应异常的 try-catch 组合。例如，在程序清单 15.12 中，当函数 hmeans( )引发异常时，控制权将传递给函数 means( )；然而，当 gmean( )引发异常时，控制权将向上传递到 main( )。

另一个不同之处是，引发异常时编译器总是创建一个临时拷贝，即使异常规范和 catch 块中指定的是引用。例如，请看下面的代码：

```
class problem {...};
...
void super() throw (problem)
{
    ...
    if (oh_no)
    {
        problem oops;   // construct object
        throw oops;     // throw it
    ...
}
...
try {
    super();
}
catch(problem & p)
{
// statements
}
```

p 将指向 oops 的副本而不是 oops 本身。这是件好事，因为函数 super( )执行完毕后，oops 将不复存在。顺便说一句，将引发异常和创建对象组合在一起将更简单：

```
throw problem();     // construct and throw default problem object
```

您可能会问，既然 throw 语句将生成副本，为何代码中使用引用呢？毕竟，将引用作为返回值的通常原因是避免创建副本以提高效率。答案是，引用还有另一个重要特征：基类引用可以执行派生类对象。假设有一组通过继承关联起来的异常类型，则在异常规范中只需列出一个基类引用，它将与任何派生类对象匹配。

假设有一个异常类层次结构，并要分别处理不同的异常类型，则使用基类引用将能够捕获任何异常对象；而使用派生类对象只能捕获它所属类及从这个类派生而来的类的对象。引发的异常对象将被第一个与之匹配的 catch 块捕获。这意味着 catch 块的排列顺序应该与派生顺序相反：

```
class bad_1 {...};
class bad_2 : public bad_1 {...};
class bad_3 : public bad_2 {...};
...
void duper()
{
    ...
    if (oh_no)
        throw bad_1();
    if (rats)
        throw bad_2();
    if (drat)
        throw bad_3();
}
...
try {
    duper();
}
catch(bad_3 &be)
{ // statements }
catch(bad_2 &be)
{ // statements }
catch(bad_1 &be)
{ // statements }
```

如果将 bad_1 &处理程序放在最前面，它将捕获异常 bad_1、bad_2 和 bad_3；通过按相反的顺序排列，bad_3 异常将被 bad_3 &处理程序所捕获。

提示：如果有一个异常类继承层次结构，应这样排列 catch 块：将捕获位于层次结构最下面的异常类的 catch 语句放在最前面，将捕获基类异常的 catch 语句放在最后面。

通过正确地排列 catch 块的顺序，让您能够在如何处理异常方面有选择的余地。然而，有时候可能不知道会发生哪些异常。例如，假设您编写了一个调用另一个函数的函数，而您并不知道被调用的函数可能引发哪些异常。在这种情况下，仍能够捕获异常，即使不知道异常的类型。方法是使用省略号来表示异常类型，从而捕获任何异常：

```
catch (...) { // statements }  // catches any type exception
```

如果知道一些可能会引发的异常，可以将上述捕获所有异常的 catch 块放在最后面，这有点类似于 switch 语句中的 default：

```
try {
    duper();
}
catch(bad_3 &be)
{ // statements }
catch(bad_2 &be)
{ // statements }
catch(bad_1 &be)
{ // statements }
```

```
catch(bad_hmean & h)
{ // statements }
catch (...)          // catch whatever is left
{ // statements }
```

可以创建捕获对象而不是引用的处理程序。在 catch 语句中使用基类对象时，将捕获所有的派生类对象，但派生特性将被剥去，因此将使用虚方法的基类版本。

15.3.8　exception 类

C++异常的主要目的是为设计容错程序提供语言级支持，即异常使得在程序设计中包含错误处理功能更容易，以免事后采取一些严格的错误处理方式。异常的灵活性和相对方便性激励着程序员在条件允许的情况下在程序设计中加入错误处理功能。总之，异常是这样一种特性：类似于类，可以改变您的编程方式。

较新的 C++编译器将异常合并到语言中。例如，为支持该语言，exception 头文件（以前为 exception.h 或 except.h）定义了 exception 类，C++可以把它用作其他异常类的基类。代码可以引发 exception 异常，也可以将 exception 类用作基类。有一个名为 what( )的虚拟成员函数，它返回一个字符串，该字符串的特征随实现而异。然而，由于这是一个虚方法，因此可以在从 exception 派生而来的类中重新定义它：

```
#include <exception>
class bad_hmean : public std::exception
{
public:
    const char * what() { return "bad arguments to hmean()"; }
...
};
class bad_gmean : public std::exception
{
public:
    const char * what() { return "bad arguments to gmean()"; }
...
};
```

如果不想以不同的方式处理这些派生而来的异常，可以在同一个基类处理程序中捕获它们：

```
try {
...
}
catch(std::exception & e)
{
    cout << e.what() << endl;
...
}
```

否则，可以分别捕获它们。

C++库定义了很多基于 exception 的异常类型。

1. stdexcept 异常类

头文件 stdexcept 定义了其他几个异常类。首先，该文件定义了 logic_error 和 runtime_error 类，它们都是以公有方式从 exception 派生而来的：

```
class logic_error : public exception {
public:
explicit logic_error(const string& what_arg);
...
};
```

```
class domain_error : public logic_error {
public:
explicit domain_error(const string& what_arg);
...
};
```

注意，这些类的构造函数接受一个 string 对象作为参数，该参数提供了方法 what( )以 C-风格字符串方式返回的字符数据。

这两个新类被用作两个派生类系列的基类。异常类系列 logic_error 描述了典型的逻辑错误。总体而言，通过合理的编程可以避免这种错误，但实际上这些错误还是可能发生的。每个类的名称指出了它用于报告的错误类型：

- domain_error;
- invalid_argument;
- length_error;
- out_of_bounds。

每个类独有一个类似于 logic_error 的构造函数，让您能够提供一个供方法 what( )返回的字符串。

数学函数有定义域（domain）和值域（range）。定义域由参数的可能取值组成，值域由函数可能的返回值组成。例如，正弦函数的定义域为负无穷大到正无穷大，因为任何实数都有正弦值；但正弦函数的值域为−1 到+1，因为它们分别是最大和最小正弦值。另一方面，反正弦函数的定义域为−1 到+1，值域为−π 到+ π。如果您编写一个函数，该函数将一个参数传递给函数 std::sin( )，则可以让该函数在参数不在定义域−1 到+1 之间时引发 domain_error 异常。

异常 invalid_argument 指出给函数传递了一个意料外的值。例如，如果函数希望接受一个这样的字符串：其中每个字符要么是‘0’要么是‘1’，则当传递的字符串中包含其他字符时，该函数将引发 invalid_argument 异常。

异常 length_error 用于指出没有足够的空间来执行所需的操作。例如，string 类的 append( )方法在合并得到的字符串长度超过最大允许长度时，将引发 length_error 异常。

异常 out_of_bounds 通常用于指示索引错误。例如，您可以定义一个类似于数组的类，其 operator( ) [ ] 在使用的索引无效时引发 out_of_bounds 异常。

接下来，runtime_error 异常系列描述了可能在运行期间发生但难以预计和防范的错误。每个类的名称指出了它用于报告的错误类型：

- range_error;
- overflow_error;
- underflow_error。

每个类独有一个类似于 runtime_error 的构造函数，让您能够提供一个供方法 what( )返回的字符串。

下溢（underflow）错误在浮点数计算中。一般而言，存在浮点类型可以表示的最小非零值，计算结果比这个值还小时将导致下溢错误。整型和浮点型都可能发生上溢错误，当计算结果超过了某种类型能够表示的最大数量级时，将发生上溢错误。计算结果可能不再函数允许的范围之内，但没有发生上溢或下溢错误，在这种情况下，可以使用 range_error 异常。

一般而言，logic_error 系列异常表明存在可以通过编程修复的问题，而 runtime_error 系列异常表明存在无法避免的问题。所有这些错误类有相同的常规特征，它们之间的主要区别在于：不同的类名让您能够分别处理每种异常。另一方面，继承关系让您能够一起处理它们（如果您愿意的话）。例如，下面的代码首先单独捕获 out_of_bounds 异常，然后统一捕获其他 logic_error 系列异常，最后统一捕获 exception 异常、runtime_error 系列异常以及其他从 exception 派生而来的异常：

```
try {
...
}
catch(out_of_bounds & oe) // catch out_of_bounds error
{...}
```

```
catch(logic_error & oe)    // catch remaining logic_error family
{...}
catch(exception & oe)      // catch runtime_error, exception objects
{...}
```

如果上述库类不能满足您的需求，应该从 logic_error 或 runtime_error 派生一个异常类，以确保您异常类可归入同一个继承层次结构中。

2. bad_alloc 异常和 new

对于使用 new 导致的内存分配问题，C++的最新处理方式是让 new 引发 bad_alloc 异常。头文件 new 包含 bad_alloc 类的声明，它是从 exception 类公有派生而来的。但在以前，当无法分配请求的内存量时，new 返回一个空指针。

程序清单 15.13 演示了最新的方法。捕获到异常后，程序将显示继承的 what( )方法返回的消息（该消息随实现而异），然后终止。

程序清单 15.13　newexcp.cpp

```
// newexcp.cpp -- the bad_alloc exception
#include <iostream>
#include <new>
#include <cstdlib>  // for exit(), EXIT_FAILURE
using namespace std;

struct Big
{
    double stuff[20000];
};

int main()
{
    Big * pb;
    try {
        cout << "Trying to get a big block of memory:\n";
        pb = new Big[10000]; // 1,600,000,000 bytes
        cout << "Got past the new request:\n";
    }
    catch (bad_alloc & ba)
    {
        cout << "Caught the exception!\n";
        cout << ba.what() << endl;
        exit(EXIT_FAILURE);
    }
    cout << "Memory successfully allocated\n";
    pb[0].stuff[0] = 4;
    cout << pb[0].stuff[0] << endl;
    delete [] pb;
    return 0;
}
```

下面该程序在某个系统中的输出：

```
Trying to get a big block of memory:
Caught the exception!
std::bad_alloc
```

在这里，方法 what( )返回字符串“std::bad_alloc”。

如果程序在您的系统上运行时没有出现内存分配问题，可尝试提高请求分配的内存量。

3. 空指针和 new

很多代码都是在 new 在失败时返回空指针时编写的。为处理 new 的变化，有些编译器提供了一个标记（开关），让用户选择所需的行为。当前，C++标准提供了一种在失败时返回空指针的 new，其用法如下：

```
int * pi = new (std::nothrow) int;
int * pa = new (std::nowthrow) int[500];
```

使用这种 new，可将程序清单 15.13 的核心代码改为如下所示：

```
Big * pb;

pb = new (std::nothrow) Big[10000]; // 1,600,000,000 bytes
if (pb == 0)
{
    cout << "Could not allocate memory. Bye.\n";
    exit(EXIT_FAILURE);
}
```

15.3.9 异常、类和继承

异常、类和继承以三种方式相互关联。首先，可以像标准 C++库所做的那样，从一个异常类派生出另一个；其次，可以在类定义中嵌套异常类声明来组合异常；第三，这种嵌套声明本身可被继承，还可用作基类。

程序清单 15.14 带领我们开始了上述一些可能性的探索之旅。这个头文件声明了一个 Sales 类，它用于存储一个年份以及一个包含 12 个月的销售数据的数组。LabeledSales 类是从 Sales 派生而来的，新增了一个用于存储数据标签的成员。

程序清单 15.14　sales.h

```
// sales.h  -- exceptions and inheritance
#include <stdexcept>
#include <string>

class Sales
{
public:
    enum {MONTHS = 12};   // could be a static const
    class bad_index : public std::logic_error
    {
    private:
        int bi;  // bad index value
    public:
        explicit bad_index(int ix,
            const std::string & s = "Index error in Sales object\n");
        int bi_val() const {return bi;}
        virtual ~bad_index() throw() {}
    };
    explicit Sales(int yy = 0);
    Sales(int yy, const double * gr, int n);
    virtual ~Sales() { }
    int Year() const { return year; }
    virtual double operator[](int i) const;
    virtual double & operator[](int i);
private:
    double gross[MONTHS];
```

```
    int year;
};

class LabeledSales : public Sales
{
  public:
    class nbad_index : public Sales::bad_index
    {
    private:
        std::string lbl;
    public:
        nbad_index(const std::string & lb, int ix,
           const std::string & s = "Index error in LabeledSales object\n");
        const std::string & label_val() const {return lbl;}
        virtual ~nbad_index() throw() {}
    };
    explicit LabeledSales(const std::string & lb = "none", int yy = 0);
    LabeledSales(const std::string & lb, int yy, const double * gr, int n);
    virtual ~LabeledSales() { }
    const std::string & Label() const {return label;}
    virtual double operator[](int i) const;
    virtual double & operator[](int i);
private:
    std::string label;
};
```

来看一下程序清单 15.14 的几个细节。首先，符号常量 MONTHS 位于 Sales 类的保护部分，这使得派生类（如 LabeledSales）能够使用这个值。

接下来，bad_index 被嵌套在 Sales 类的公有部分中，这使得客户类的 catch 块可以使用这个类作为类型。注意，在外部使用这个类型时，需要使用 Sales::bad_index 来标识。这个类是从 logic_error 类派生而来的，能够存储和报告数组索引的超界值（out-of-bounds value）。

nbad_index 类被嵌套到 LabeledSales 的公有部分，这使得客户类可以通过 LabeledSales::nbad_index 来使用它。它是从 bad_index 类派生而来的，新增了存储和报告 LabeledSales 对象的标签的功能。由于 bad_index 是从 logic_error 派生而来的，因此 nbad_index 归根结底也是从 logic_error 派生而来的。

这两个类都有重载的 operator[ ] ( )方法，这些方法设计用于访问存储在对象中的数组元素，并在索引超界时引发异常。

bad_index 和 nbad_index 类都使用了异常规范 throw()，这是因为它们都归根结底是从基类 exception 派生而来的，而 exception 的虚构造函数使用了异常规范 throw()。这是 C++98 的一项功能，在 C++11 中，exception 的构造函数没有使用异常规范。

程序清单 15.15 是程序清单中没有声明为内联的方法的实现。注意，对于被嵌套类的方法，需要使用多个作用域解析运算符。另外，如果数组索引超界，函数 operator[ ] ( )将引发异常。

程序清单 15.15　sales.cpp

```
// sales.cpp -- Sales implementation
#include "sales.h"
using std::string;

Sales::bad_index::bad_index(int ix, const string & s )
    : std::logic_error(s), bi(ix)
{
}
```

```
Sales::Sales(int yy)
{
    year = yy;
    for (int i = 0; i < MONTHS; ++i)
        gross[i] = 0;
}

Sales::Sales(int yy, const double * gr, int n)
{
    year = yy;
    int lim = (n < MONTHS)? n : MONTHS;
    int i;
    for (i = 0; i < lim; ++i)
        gross[i] = gr[i];
    // for i > n and i < MONTHS
    for ( ; i < MONTHS; ++i)
        gross[i] = 0;
}

double Sales::operator[](int i) const
{
    if(i < 0 || i >= MONTHS)
        throw bad_index(i);
    return gross[i];
}

double & Sales::operator[](int i)
{
    if(i < 0 || i >= MONTHS)
        throw bad_index(i);
    return gross[i];
}

LabeledSales::nbad_index::nbad_index(const string & lb, int ix,
          const string & s ) : Sales::bad_index(ix, s)
{
    lbl = lb;
}

LabeledSales::LabeledSales(const string & lb, int yy)
        : Sales(yy)
{
    label = lb;
}

LabeledSales::LabeledSales(const string & lb, int yy,
                           const double * gr, int n)
                                  : Sales(yy, gr, n)
{
    label = lb;
}

double LabeledSales::operator[](int i) const
{    if(i < 0 || i >= MONTHS)
        throw nbad_index(Label(), i);
```

```
    return Sales::operator[](i);
}

double & LabeledSales::operator[](int i)
{
    if(i < 0 || i >= MONTHS)
        throw nbad_index(Label(), i);
    return Sales::operator[](i);
}
```

程序清单 15.16 在一个程序中使用了这些类：首先试图超越 LabeledSales 对象 sales2 中数组的末尾，然后试图超越 Sales 对象 sales1 中数组的末尾。这些尝试是在两个 try 块中进行的，让您能够检测每种异常。

程序清单 15.16　use_sales.cpp

```
// use_sales.cpp  -- nested exceptions
#include <iostream>
#include "sales.h"

int main()
{
    using std::cout;
    using std::cin;
    using std::endl;
 double vals1[12] =
 {
     1220, 1100, 1122, 2212, 1232, 2334,
     2884, 2393, 3302, 2922, 3002, 3544
 };

 double vals2[12] =
 {
     12, 11, 22, 21, 32, 34,
     28, 29, 33, 29, 32, 35
 };

 Sales sales1(2011, vals1, 12);
 LabeledSales sales2("Blogstar",2012, vals2, 12 );

 cout << "First try block:\n";
 try
 {
     int i;
     cout << "Year = " << sales1.Year() << endl;
     for (i = 0; i < 12; ++i)
     {

         cout << sales1[i] << ' ';
         if (i % 6 == 5)
             cout << endl;
     }
     cout << "Year = " << sales2.Year() << endl;
     cout << "Label = " << sales2.Label() << endl;
     for (i = 0; i <= 12; ++i)
     {
```

```
            cout << sales2[i] << ' ';
            if (i % 6 == 5)
                cout << endl;
        }
        cout << "End of try block 1.\n";
    }
    catch(LabeledSales::nbad_index & bad)
    {
        cout << bad.what();
        cout << "Company: " << bad.label_val() << endl;
        cout << "bad index: " << bad.bi_val() << endl;
    }
    catch(Sales::bad_index & bad)
    {
        cout << bad.what();
           cout << "bad index: " << bad.bi_val() << endl;
    }
    cout << "\nNext try block:\n";
    try
     {
           sales2[2] = 37.5;
           sales1[20] = 23345;
           cout << "End of try block 2.\n";
    }
    catch(LabeledSales::nbad_index & bad)
    {
           cout << bad.what();
           cout << "Company: " << bad.label_val() << endl;
           cout << "bad index: " << bad.bi_val() << endl;
    }
    catch(Sales::bad_index & bad)
    {
           cout << bad.what();
           cout << "bad index: " << bad.bi_val() << endl;
    }
    cout << "done\n";

     return 0;
}
```

下面是程序清单 15.14～程序清单 15.16 组成的程序的输出：

```
First try block:
Year = 2011
1220 1100 1122 2212 1232 2334
2884 2393 3302 2922 3002 3544
Year = 2012
Label = Blogstar
12 11 22 21 32 34
28 29 33 29 32 35
Index error in LabeledSales object
Company: Blogstar
bad index: 12

Next try block:
```

```
Index error in Sales object
bad index: 20
done
```

15.3.10　异常何时会迷失方向

异常被引发后，在两种情况下，会导致问题。首先，如果它是在带异常规范的函数中引发的，则必须与规范列表中的某种异常匹配（在继承层次结构中，类类型与这个类及其派生类的对象匹配），否则称为意外异常（unexpected exception）。在默认情况下，这将导致程序异常终止（虽然 C++11 摒弃了异常规范，但仍支持它，且有些现有的代码使用了它）。如果异常不是在函数中引发的（或者函数没有异常规范），则必须捕获它。如果没被捕获（在没有 try 块或没有匹配的 catch 块时，将出现这种情况），则异常被称为未捕获异常（uncaught exception）。在默认情况下，这将导致程序异常终止。然而，可以修改程序对意外异常和未捕获异常的反应。下面来看如何修改，先从未捕获异常开始。

未捕获异常不会导致程序立刻异常终止。相反，程序将首先调用函数 terminate( )。在默认情况下，terminate( )调用 abort( )函数。可以指定 terminate( )应调用的函数（而不是 abort( )）来修改 terminate( )的这种行为。为此，可调用 set_terminate( )函数。set_terminate( )和 terminate( )都是在头文件 exception 中声明的：

```
typedef void (*terminate_handler)();
terminate_handler set_terminate(terminate_handler f) throw();  // C++98
terminate_handler set_terminate(terminate_handler f) noexcept; // C++11
void terminate();              // C++98
void terminate() noexcept;     // C++11
```

其中的 typedef 使 terminate_handler 成为这样一种类型的名称：指向没有参数和返回值的函数的指针。set_terminate( )函数将不带任何参数且返回类型为 void 的函数的名称（地址）作为参数，并返回该函数的地址。如果调用了 set_terminate( )函数多次，则 terminate( )将调用最后一次 set_terminate( )调用设置的函数。

来看一个例子。假设希望未捕获的异常导致程序打印一条消息，然后调用 exit( )函数，将退出状态值设置为 5。首先，请包含头文件 exception。可以使用 using 编译指令、适当的 using 声明或 std ::限定符，来使其声明可用。

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

然后，设计一个完成上述两种操作所需的函数，其原型如下：

```
void myQuit()
{
    cout << "Terminating due to uncaught exception\n";
    exit(5);
}
```

最后，在程序的开头，将终止操作指定为调用该函数。

```
set_terminate(myQuit);
```

现在，如果引发了一个异常且没有被捕获，程序将调用 terminate( )，而后者将调用 MyQuit( )。

接下来看意外异常。通过给函数指定异常规范，可以让函数的用户知道要捕获哪些异常。假设函数的原型如下：

```
double Argh(double, double) throw(out_of_bounds);
```

则可以这样使用该函数：

```
try {
    x = Argh(a, b);
}
catch(out_of_bounds & ex)
{
```

```
    ...
}
```

知道应捕获哪些异常很有帮助，因为默认情况下，未捕获的异常将导致程序异常终止。

原则上，异常规范应包含函数调用的其他函数引发的异常。例如，如果 Argh( )调用了 Duh( )函数，而后者可能引发 retort 对象异常，则 Argh( )和 Duh( )的异常规范中都应包含 retort。除非自己编写所有的函数，并且特别仔细，否则无法保证上述工作都已正确完成。例如，可能使用的是老式商业库，而其中的函数没有异常规范。这表明应进一步探讨这样一点，即如果函数引发了其异常规范中没有的异常，情况将如何？这也表明异常规范机制处理起来比较麻烦，这也是 C++11 将其摒弃的原因之一。

在这种情况下，行为与未捕获的异常极其类似。如果发生意外异常，程序将调用 unexpected( )函数（您没有想到是 unexpected( )函数吧？谁也想不到！）。这个函数将调用 terminate( )，后者在默认情况下将调用 abort( )。正如有一个可用于修改 terminate( )的行为的 set_terminate( )函数一样，也有一个可用于修改 unexpected( )的行为的 set_unexpected( )函数。这些新函数也是在头文件 exception 中声明的：

```
typedef void (*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler f) throw();   // C++98
unexpected_handler set_unexpected(unexpected_handler f) noexcept;  // C++11
void unexpected();            // C++98
void unexpected() noexcept;   // C+0x
```

然而，与提供给 set_terminate( )的函数的行为相比，提供给 set_unexpected( )的函数的行为受到更严格的限制。具体地说，unexpected_handler 函数可以：

- 通过调用 terminate( )（默认行为）、abort( )或 exit( )来终止程序；
- 引发异常。

引发异常（第二种选择）的结果取决于 unexpected_handler 函数所引发的异常以及引发意外异常的函数的异常规范：

- 如果新引发的异常与原来的异常规范匹配，则程序将从那里开始进行正常处理，即寻找与新引发的异常匹配的 catch 块。基本上，这种方法将用预期的异常取代意外异常；
- 如果新引发的异常与原来的异常规范不匹配，且异常规范中没有包括 std ::bad_exception 类型，则程序将调用 terminate( )。bad_exception 是从 exception 派生而来的，其声明位于头文件 exce ption 中；
- 如果新引发的异常与原来的异常规范不匹配，且原来的异常规范中包含了 std ::bad_exception 类型，则不匹配的异常将被 std ::bad_exception 异常所取代。

总之，如果要捕获所有的异常（不管是预期的异常还是意外异常），则可以这样做：

首先确保异常头文件的声明可用：

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

然后，设计一个替代函数，将意外异常转换为 bad_exception 异常，该函数的原型如下：

```
void myUnexpected()
{
    throw std::bad_exception();  //or just throw;
}
```

仅使用 throw，而不指定异常将导致重新引发原来的异常。然而，如果异常规范中包含了这种类型，则该异常将被 bad_exception 对象所取代。

接下来在程序的开始位置，将意外异常操作指定为调用该函数：

```
set_unexpected(myUnexpected);
```

最后，将 bad_exception 类型包括在异常规范中，并添加如下 catch 块序列：

```
double Argh(double, double) throw(out_of_bounds, bad_exception);
...
try {
    x = Argh(a, b);
}
catch(out_of_bounds & ex)
{
    ...
}
catch(bad_exception & ex)
{
    ...
}
```

15.3.11　有关异常的注意事项

从前面关于如何使用异常的讨论可知，应在设计程序时就加入异常处理功能，而不是以后再添加。这样做有些缺点。例如，使用异常会增加程序代码，降低程序的运行速度。异常规范不适用于模板，因为模板函数引发的异常可能随特定的具体化而异。异常和动态内存分配并非总能协同工作。

下面进一步讨论动态内存分配和异常。首先，请看下面的函数：

```
void test1(int n)
{
    string mesg("I'm trapped in an endless loop");
    ...
    if (oh_no)
        throw exception();
    ...
    return;
}
```

string 类采用动态内存分配。通常，当函数结束时，将为 mesg 调用 string 的析构函数。虽然 throw 语句过早地终止了函数，但它仍然使得析构函数被调用，这要归功于栈解退。因此在这里，内存被正确地管理。

接下来看下面这个函数：

```
void test2(int n)
{
    double * ar = new double[n];
    ...
    if (oh_no)
        throw exception();
    ...
    delete [] ar;
    return;
}
```

这里有个问题。解退栈时，将删除栈中的变量 ar。但函数过早的终止意味着函数末尾的 delete[ ]语句被忽略。指针消失了，但它指向的内存块未被释放，并且不可访问。总之，这些内存被泄漏了。

这种泄漏是可以避免的。例如，可以在引发异常的函数中捕获该异常，在 catch 块中包含一些清理代码，然后重新引发异常：

```
void test3(int n)
{
    double * ar = new double[n];
    ...
```

```
    try {
        if (oh_no)
            throw exception();
    }
    catch(exception & ex)
    {
        delete [] ar;
        throw;
    }
    ...
    delete [] ar;
    return;
}
```

然而，这将增加疏忽和产生其他错误的机会。另一种解决方法是使用第 16 章将讨论的智能指针模板之一。

总之，虽然异常处理对于某些项目极为重要，但它也会增加编程的工作量、增大程序、降低程序的速度。另一方面，不进行错误检查的代价可能非常高。

异 常 处 理

在现代库中，异常处理的复杂程度可能再创新高——主要原因在于文档没有对异常处理例程进行解释或解释得很蹩脚。任何熟练使用现代操作系统的人都遇到过未处理的异常导致的错误和问题。这些错误背后的程序员通常面临一场艰难的战役，需要不断了解库的复杂性：什么异常将被引发，它们发生的原因和时间，如何处理它们，等等。

程序员新手很快将发现，理解库中异常处理像学习语言本身一样困难，现代库中包含的例程和模式可能像 C++语法细节一样陌生而困难。要开发出优秀的软件，必须花时间了解库和类中的复杂内容，就像必须花时间学习 C++本身一样。通过库文档和源代码了解到的异常和错误处理细节将使程序员和他的软件受益。

15.4 RTTI

RTTI 是运行阶段类型识别（Runtime Type Identification）的简称。这是新添加到 C++中的特性之一，很多老式实现不支持。另一些实现可能包含开关 RTTI 的编译器设置。RTTI 旨在为程序在运行阶段确定对象的类型提供一种标准方式。很多类库已经为其类对象提供了实现这种功能的方式，但由于 C++内部并不支持，因此各个厂商的机制通常互不兼容。创建一种 RTTI 语言标准将使得未来的库能够彼此兼容。

15.4.1 RTTI 的用途

假设有一个类层次结构，其中的类都是从同一个基类派生而来的，则可以让基类指针指向其中任何一个类的对象。这样便可以调用这样的函数：在处理一些信息后，选择一个类，并创建这种类型的对象，然后返回它的地址，而该地址可以被赋给基类指针。如何知道指针指向的是哪种对象呢？

在回答这个问题之前，先考虑为何要知道类型。可能希望调用类方法的正确版本，在这种情况下，只要该函数是类层次结构中所有成员都拥有的虚函数，则并不真正需要知道对象的类型。但派生对象可能包含不是继承而来的方法，在这种情况下，只有某些类型的对象可以使用该方法。也可能是出于调试目的，想跟踪生成的对象的类型。对于后两种情况，RTTI 提供解决方案。

15.4.2 RTTI 的工作原理

C++有 3 个支持 RTTI 的元素。

- 如果可能的话，dynamic_cast 运算符将使用一个指向基类的指针来生成一个指向派生类的指针；否则，该运算符返回 0——空指针。
- typeid 运算符返回一个指出对象的类型的值。
- type_info 结构存储了有关特定类型的信息。

只能将 RTTI 用于包含虚函数的类层次结构，原因在于只有对于这种类层次结构，才应该将派生对象的地址赋给基类指针。

警告：RTTI 只适用于包含虚函数的类。

下面详细介绍 RTTI 的这 3 个元素。

1. dynamic_cast 运算符

dynamic_cast 运算符是最常用的 RTTI 组件，它不能回答“指针指向的是哪类对象”这样的问题，但能够回答“是否可以安全地将对象的地址赋给特定类型的指针”这样的问题。我们来看一看这意味着什么。假设有下面这样的类层次结构：

```
class Grand { // has virtual methods};
class Superb : public Grand { ... };
class Magnificent : public Superb { ... };
```

接下来假设有下面的指针：

```
Grand * pg = new Grand;
Grand * ps = new Superb;
Grand * pm = new Magnificent;
```

最后，对于下面的类型转换：

```
Magnificent * p1 = (Magnificent *) pm;        // #1
Magnificent * p2 = (Magnificent *) pg;        // #2
Superb * p3 = (Magnificent *) pm;             // #3
```

哪些是安全的？根据类声明，它们可能全都是安全的，但只有那些指针类型与对象的类型（或对象的直接或间接基类的类型）相同的类型转换才一定是安全的。例如，类型转换#1 就是安全的，因为它将 Magificent 类型的指针指向类型为 Magnificent 的对象。类型转换#2 就是不安全的，因为它将基数对象（Grand）的地址赋给派生类（Magnificent）指针。因此，程序将期望基类对象有派生类的特征，而通常这是不可能的。例如，Magnificent 对象可能包含一些 Grand 对象没有的数据成员。然而，类型转换#3 是安全的，因为它将派生对象的地址赋给基类指针。即公有派生确保 Magnificent 对象同时也是一个 Superb 对象（直接基类）和一个 Grand 对象（间接基类）。因此，将它的地址赋给这 3 种类型的指针都是安全的。虚函数确保了将这 3 种指针中的任何一种指向 Magnificent 对象时，都将调用 Magnificent 方法。

注意，与问题“指针指向的是哪种类型的对象”相比，问题“类型转换是否安全”更通用，也更有用。通常想知道类型的原因在于：知道类型后，就可以知道调用特定的方法是否安全。要调用方法，类型并不一定要完全匹配，而可以是定义了方法的虚拟版本的基类类型。下面的例子说明了这一点。

然而，先来看一下 dynamic_cast 的语法。该运算符的用法如下，其中 pg 指向一个对象：

```
Superb * pm = dynamic_cast<Superb *>(pg);
```

这提出了这样的问题：指针 pg 的类型是否可被安全地转换为 Superb *？如果可以，运算符将返回对象的地址，否则返回一个空指针。

注意：通常，如果指向的对象（*pt）的类型为 Type 或者是从 Type 直接或间接派生而来的类型，则下面的表达式将指针 pt 转换为 Type 类型的指针：

```
dynamic_cast<Type *>(pt)
```

否则，结果为 0，即空指针。

程序清单 15.17 演示了这种处理。首先，它定义了 3 个类，名称为 Grand、Superb 和 Magnificent。Grand 类定义了一个虚函数 Speak( )，而其他类都重新定义了该虚函数。Superb 类定义了一个虚函数 Say( )，而 Manificent 也重新定义了它（参见图 15.4）。程序定义了 GetOne( )函数，该函数随机创建这 3 种类中某种类的对象，并对其进行初始化，然后将地址作为 Grand*指针返回（GetOne( )函数模拟用户做出决定）。循环将该指针赋给 Grand *变量 pg，然后使用 pg 调用 Speak( )函数。因为这个函数是虚拟的，所以代码能够正确地调用指向的对象的 Speak( )版本。

```
for (int i = 0; i < 5; i++)
{
    pg = GetOne();
    pg->Speak();
    ...
}
```

然而，不能用相同的方式（即使用指向 Grand 的指针）来调用 Say( )函数，因为 Grand 类没有定义它。然而，可以使用 dynamic_cast 运算符来检查是否可将 pg 的类型安全地转换为 Superb 指针。如果对象的类型为 Superb 或 Magnificent，则可以安全转换。在这两种情况下，都可以安全地调用 Say( )函数：

```
if (ps = dynamic_cast<Superb *>(pg))
    ps->Say();
```

赋值表达式的值是它左边的值，因此 if 条件的值为 ps。如果类型转换成功，则 ps 的值为非零（true）；如果类型转换失败，即 pg 指向的是一个 Grand 对象，ps 的值将为 0（false）。程序清单 15.17 列出了所有的代码。顺便说一句，有些编译器可能会对无目的赋值（在 if 条件语句中，通常使用= =运算符）提出警告。

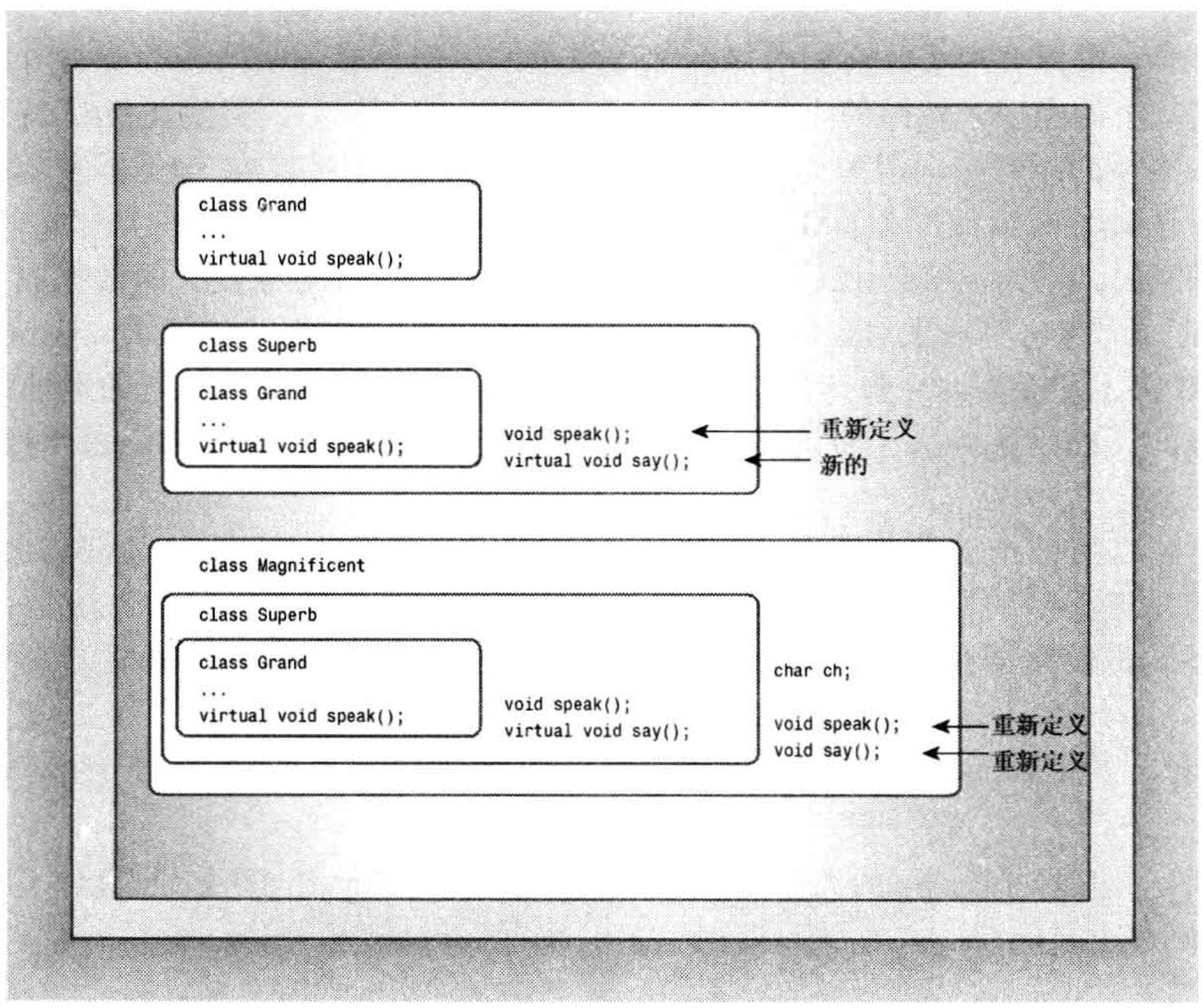

图 15.4 Grand 类系列

程序清单 15.17　rtti1.cpp

```
// rtti1.cpp -- using the RTTI dynamic_cast operator
#include <iostream>
#include <cstdlib>
#include <ctime>

using std::cout;
class Grand
{
private:
    int hold;
public:
    Grand(int h = 0) : hold(h) {}
    virtual void Speak() const { cout << "I am a grand class!\n";}
    virtual int Value() const { return hold; }
};

class Superb : public Grand
{
public:
    Superb(int h = 0) : Grand(h) {}
    void Speak() const {cout << "I am a superb class!!\n"; }
    virtual void Say() const
        { cout << "I hold the superb value of " << Value() << "!\n";}
};

class Magnificent : public Superb
{
private:
    char ch;
public:
    Magnificent(int h = 0, char c = 'A') : Superb(h), ch(c) {}
    void Speak() const {cout << "I am a magnificent class!!!\n";}
    void Say() const {cout << "I hold the character " << ch <<
                " and the integer "  << Value() << "!\n"; }
};

Grand * GetOne();

int main()
{
    std::srand(std::time(0));
    Grand * pg;
    Superb * ps;
    for (int i = 0; i < 5; i++)
    {
        pg = GetOne();
        pg->Speak();
        if( ps = dynamic_cast<Superb *>(pg))
            ps->Say();
    }
    return 0;
}
```

```
Grand * GetOne()    // generate one of three kinds of objects randomly
{
    Grand * p;
    switch( std::rand() % 3)
    {
        case 0: p = new Grand(std::rand() % 100);
                    break;
        case 1: p = new Superb(std::rand() % 100);
                    break;
        case 2: p = new Magnificent(std::rand() % 100,
                                'A' + std::rand() % 26);
                    break;
    }
    return p;
}
```

注意：即使编译器支持 RTTI，在默认情况下，它也可能关闭该特性。如果该特性被关闭，程序可能仍能够通过编译，但将出现运行阶段错误。在这种情况下，您应查看文档或菜单选项。

程序清单 15.17 中程序说明了重要的一点，即应尽可能使用虚函数，而只在必要时使用 RTTI。下面是该程序的输出：

```
I am a superb class!!
I hold the superb value of 68!
I am a magnificent class!!!
I hold the character R and the integer 68!
I am a magnificent class!!!
I hold the character D and the integer 12!
I am a magnificent class!!!
I hold the character V and the integer 59!
I am a grand class!
```

正如您看到的，只为 Superb 和 Magnificent 类调用了 Say( )方法（每次运行时输出都可能不同，因为该程序使用 rand( )来选择对象类型）。

也可以将 dynamic_cast 用于引用，其用法稍微有点不同：没有与空指针对应的引用值，因此无法使用特殊的引用值来指示失败。当请求不正确时，dynamic_cast 将引发类型为 bad_cast 的异常，这种异常是从 exception 类派生而来的，它是在头文件 typeinfo 中定义的。因此，可以像下面这样使用该运算符，其中 rg 是对 Grand 对象的引用：

```
#include <typeinfo> // for bad_cast
...
try {
    Superb & rs = dynamic_cast<Superb &>(rg);
    ...
}
catch(bad_cast &){
    ...
};
```

2. typeid 运算符和 type_info 类

typeid 运算符使得能够确定两个对象是否为同种类型。它与 sizeof 有些相像，可以接受两种参数：

- 类名；
- 结果为对象的表达式。

typeid 运算符返回一个对 type_info 对象的引用，其中，type_info 是在头文件 typeinfo（以前为 typeinfo.h）

中定义的一个类。type_info 类重载了==和!=运算符，以便可以使用这些运算符来对类型进行比较。例如，如果 pg 指向的是一个 Magnificent 对象，则下述表达式的结果为 bool 值 true，否则为 false：

```
typeid(Magnificent) == typeid(*pg)
```

如果 pg 是一个空指针，程序将引发 bad_typeid 异常。该异常类型是从 exception 类派生而来的，是在头文件 typeinfo 中声明的。

type_info 类的实现随厂商而异，但包含一个 name( )成员，该函数返回一个随实现而异的字符串：通常（但并非一定）是类的名称。例如，下面的语句显示指针 pg 指向的对象所属的类定义的字符串：

```
cout << "Now processing type " << typeid(*pg).name() << ".\n";
```

程序清单 15.18 对程序清单 15.17 作了修改，以使用 typeid 运算符和 name( )成员函数。注意，它们都适用于 dynamic_cast 和 virtual 函数不能处理的情况。typeid 测试用来选择一种操作，因为操作不是类的方法，所以不能通过类指针调用它。name( )方法语句演示了如何将方法用于调试。注意，程序包含了头文件 typeinfo。

程序清单 15.18　rtti2.cpp

```
// rtti2.cpp  -- using dynamic_cast, typeid, and type_info
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <typeinfo>
using namespace std;
class Grand
{
private:
    int hold;
public:
    Grand(int h = 0) : hold(h) {}
    virtual void Speak() const { cout << "I am a grand class!\n";}
    virtual int Value() const { return hold; }
};

class Superb : public Grand
{
public:
    Superb(int h = 0) : Grand(h) {}
    void Speak() const {cout << "I am a superb class!!\n"; }
    virtual void Say() const
        { cout << "I hold the superb value of " << Value() << "!\n";}
};

class Magnificent : public Superb
{
private:
    char ch;
public:
    Magnificent(int h = 0, char cv = 'A') : Superb(h), ch(cv) {}
    void Speak() const {cout << "I am a magnificent class!!!\n";}
    void Say() const {cout << "I hold the character " << ch <<
                " and the integer "  << Value() << "!\n"; }
};

Grand * GetOne();
int main()
```

```
{
    srand(time(0));
    Grand * pg;
    Superb * ps;
    for (int i = 0; i < 5; i++)
    {
        pg = GetOne();
        cout << "Now processing type " << typeid(*pg).name() << ".\n";
        pg->Speak();
        if( ps = dynamic_cast<Superb *>(pg))
            ps->Say();
        if (typeid(Magnificent) == typeid(*pg))
            cout << "Yes, you're really magnificent.\n";
    }
    return 0;
}

Grand * GetOne()
{
    Grand * p;

    switch( rand() % 3)
    {
        case 0: p = new Grand(rand() % 100);
                    break;
        case 1: p = new Superb(rand() % 100);
                    break;
        case 2: p = new Magnificent(rand() % 100, 'A' + rand() % 26);
                    break;
    }
    return p;
}
```

程序清单 15.18 所示程序的运行情况如下：

```
Now processing type Magnificent.
I am a magnificent class!!!
I hold the character P and the integer
Yes, you're really magnificent.
Now processing type Superb.
I am a superb class!!
I hold the superb value of 37!
Now processing type Grand.
I am a grand class!
Now processing type Superb.
I am a superb class!!
I hold the superb value of 18!
Now processing type Grand.
I am a grand class!
```

与前一个程序的输出一样，每次运行该程序的输出都可能不同，因为它使用 rand( )来选择类型。另外，调用 name()时，有些编译器可能提供不同的输出，如 5Grand（而不是 Grand）。

3. 误用 RTTI 的例子

C++界有很多人对 RTTI 口诛笔伐，他们认为 RTTI 是多余的，是导致程序效率低下和糟糕编程方式的罪魁祸首。这里不讨论对 RTTI 的争论，而介绍一下应避免的编程方式。

请看程序清单 15.17 的核心代码：

```
Grand * pg;
Superb * ps;
for (int i = 0; i < 5; i++)
{
    pg = GetOne();
    pg->Speak();
    if( ps = dynamic_cast<Superb *>(pg))
         ps->Say();
}
```

通过放弃 dynamic_cast 和虚函数，而使用 typeid，可以将上述代码重新编写为：

```
Grand * pg;
Superb * ps;
Magnificent * pm;
for (int i = 0; i < 5; i++)
{
    pg = GetOne();
    if (typeid(Magnificent) == typeid(*pg))
    {
        pm = (Magnificent *) pg;
        pm->Speak();
        pm->Say();
    }
    else if (typeid(Superb) == typeid(*pg))
    {
        ps = (Superb *) pg;
        ps->Speak();
        ps->Say();
    }
    else
        pg->Speak();
}
```

上述代码不仅比原来的更难看、更长，而且显式地指定各个类存在严重的缺陷。例如，假设您发现必须从 Magnificent 类派生一个 Insufferable 类，而后者需要重新定义 Speak( )和 Say( )。使用 typeid 来显示地测试每个类型时，必须修改 for 循环的代码，添加一个 else if，但无需修改原来的版本。下面的语句适用于所有从 Grand 派生而来的类：

```
pg->Speak();
```

而下面的语句适用于所有从 Superb 派生而来的类：

```
if( ps = dynamic_cast<Superb *>(pg))
      ps->Say();
```

提示：如果发现在扩展的 if else 语句系列中使用了 typeid，则应考虑是否应该使用虚函数和 dynamic_cast。

15.5　类型转换运算符

在 C++的创始人 Bjarne Stroustrup 看来，C 语言中的类型转换运算符太过松散。例如，请看下面的代码：

```
struct Data
{
    double data[200];
};
```

```
struct Junk
{
    int junk[100];
};
Data d = {2.5e33, 3.5e-19, 20.2e32};
char * pch = (char *) (&d);    // type cast #1 - convert to string
char ch = char (&d);           // type cast #2 - convert address to a char
Junk * pj = (Junk *) (&d);     // type cast #3 - convert to Junk pointer
```

首先，上述 3 种类型转换中，哪一种有意义？除非不讲理，否则它们中没有一个是有意义的。其次，这 3 种类型转换中哪种是允许的呢？在 C 语言中都是允许的。

对于这种松散情况，Stroustrop 采取的措施是，更严格地限制允许的类型转换，并添加 4 个类型转换运算符，使转换过程更规范：

- dynamic_cast；
- const_cast；
- static_cast；
- reinterpret_cast。

可以根据目的选择一个适合的运算符，而不是使用通用的类型转换。这指出了进行类型转换的原因，并让编译器能够检查程序的行为是否与设计者想法吻合。

dynamic_cast 运算符已经在前面介绍过了。总之，假设 High 和 Low 是两个类，而 ph 和 pl 的类型分别为 High *和 Low *，则仅当 Low 是 High 的可访问基类（直接或间接）时，下面的语句才将一个 Low*指针赋给 pl：

```
pl = dynamic_cast<Low *> ph;
```

否则，该语句将空指针赋给 pl。通常，该运算符的语法如下：

```
dynamic_cast < type-name > (expression)
```

该运算符的用途是，使得能够在类层次结构中进行向上转换（由于 is-a 关系，这样的类型转换是安全的），而不允许其他转换。

const_cast 运算符用于执行只有一种用途的类型转换，即改变值为 const 或 volatile，其语法与 dynamic_cast 运算符相同：

```
const_cast < type-name > (expression)
```

如果类型的其他方面也被修改，则上述类型转换将出错。也就是说，除了 const 或 volatile 特征（有或无）可以不同外，type_name 和 expression 的类型必须相同。再次假设 High 和 Low 是两个类：

```
High bar;
const High * pbar = &bar;
    ...
High * pb = const_cast<High *> (pbar);    // valid
const Low * pl = const_cast<const Low *> (pbar);      // invalid
```

第一个类型转换使得*pb 成为一个可用于修改 bar 对象值的指针，它删除 const 标签。第二个类型转换是非法的，因为它同时尝试将类型从 const High *改为 const Low *。

提供该运算符的原因是，有时候可能需要这样一个值，它在大多数时候是常量，而有时又是可以修改的。在这种情况下，可以将这个值声明为 const，并在需要修改它的时候，使用 const_cast。这也可以通过通用类型转换来实现，但通用转换也可能同时改变类型：

```
High bar;
const High * pbar = &bar;
...
High * pb = (High *) (pbar);        // valid
Low * pl = (Low *) (pbar);          // also valid
```

由于编程时可能无意间同时改变类型和常量特征，因此使用 const_cast 运算符更安全。

const_cast 不是万能的。它可以修改指向一个值的指针，但修改 const 值的结果是不确定的。程序清单 15.19 的简单示例阐明了这一点：

程序清单 15.19　constcast.cpp

```
// constcast.cpp -- using const_cast<>
#include <iostream>
using std::cout;
using std::endl;
void change(const int * pt, int n);

int main()
{
    int pop1 = 38383;
    const int pop2 = 2000;

    cout << "pop1, pop2: " << pop1 << ", " << pop2 << endl;
    change(&pop1, -103);
    change(&pop2, -103);
    cout << "pop1, pop2: " << pop1 << ", " << pop2 << endl;
    return 0;
}

void change(const int * pt, int n)
{
    int * pc;

    pc = const_cast<int *>(pt);
    *pc += n;
}
```

const_cast 运算符可以删除 const int* pt 中的 const，使得编译器能够接受 change( )中的语句：

```
*pc += n;
```

但由于 pop2 被声明为 const，因此编译器可能禁止修改它，如下面的输出所示：

```
pop1, pop2: 38383, 2000
pop1, pop2: 38280, 2000
```

正如您看到的，调用 change( )时，修改了 pop1，但没有修改 pop2。在 chang( )中，指针被声明为 const int *，因此不能用来修改指向的 int。指针 pc 删除了 const 特征，因此可用来修改指向的值，但仅当指向的值不是 const 时才可行。因此，pc 可用于修改 pop1，但不能用于修改 pop2。

static_cast 运算符的语法与其他类型转换运算符相同：

```
static_cast < type-name > (expression)
```

仅当 type_name 可被隐式转换为 expression 所属的类型或 expression 可被隐式转换为 type_name 所属的类型时，上述转换才是合法的，否则将出错。假设 High 是 Low 的基类，而 Pond 是一个无关的类，则从 High 到 Low 的转换、从 Low 到 High 的转换都是合法的，而从 Low 到 Pond 的转换是不允许的：

```
High bar;
Low blow;
...
High * pb = static_cast<High *> (&blow);     // valid upcast
Low * pl = static_cast<Low *> (&bar);        // valid downcast
Pond * pmer = static_cast<Pond *> (&blow);   // invalid, Pond unrelated
```

第一种转换是合法的，因为向上转换可以显示地进行。第二种转换是从基类指针到派生类指针，在不进行显示类型转换的情况下，将无法进行。但由于无需进行类型转换，便可以进行另一个方向的类型转换，因此使用 static_cast 来进行向下转换是合法的。

同理，由于无需进行类型转换，枚举值就可以被转换为整型，所以可以用 static_cast 将整型转换为枚举值。同样，可以使用 static_cast 将 double 转换为 int、将 float 转换为 long 以及其他各种数值转换。

reinterpret_cast 运算符用于天生危险的类型转换。它不允许删除 const，但会执行其他令人生厌的操作。有时程序员必须做一些依赖于实现的、令人生厌的操作，使用 reinterpret_cast 运算符可以简化对这种行为的跟踪工作。该运算符的语法与另外 3 个相同：

```
reinterpret_cast < type-name > (expression)
```

下面是一个使用示例：

```
struct dat {short a; short b;};
long value = 0xA224B118;
dat * pd = reinterpret_cast< dat *> (&value);
cout << hex << pd->a;   // display first 2 bytes of value
```

通常，这样的转换适用于依赖于实现的底层编程技术，是不可移植的。例如，不同系统在存储多字节整型时，可能以不同的顺序存储其中的字节。

然而，reinterprete_cast 运算符并不支持所有的类型转换。例如，可以将指针类型转换为足以存储指针表示的整型，但不能将指针转换为更小的整型或浮点型。另一个限制是，不能将函数指针转换为数据指针，反之亦然。

在 C++中，普通类型转换也受到限制。基本上，可以执行其他类型转换可执行的操作，加上一些组合，如 static_cast 或 reinterpret_cast 后跟 const_cast，但不能执行其他转换。因此，下面的类型转换在 C 语言中是允许的，但在 C++中通常不允许，因为对于大多数 C++实现，char 类型都太小，不能存储指针：

```
char ch = char (&d);          // type cast #2 - convert address to a char
```

这些限制是合理的，如果您觉得这种限制难以忍受，可以使用 C 语言。

15.6 总结

友元使得能够为类开发更灵活的接口。类可以将其他函数、其他类和其他类的成员函数作为友元。在某些情况下，可能需要使用前向声明，需要特别注意类和方法声明的顺序，以正确地组合友元。

嵌套类是在其他类中声明的类，它有助于设计这样的助手类，即实现其他类，但不必是公有接口的组成部分。

C++异常机制为处理拙劣的编程事件，如不适当的值、I/O 失败等，提供了一种灵活的方式。引发异常将终止当前执行的函数，将控制权传给匹配的 catch 块。catch 块紧跟在 try 块的后面，为捕获异常，直接或间接导致异常的函数调用必须位于 try 块中。这样程序将执行 catch 块中的代码。这些代码试图解决问题或终止程序。类可以包含嵌套的异常类，嵌套异常类在相应的问题被发现时将被引发。函数可以包含异常规范，指出在该函数中可能引发的异常；但 C++11 摒弃了这项功能。未被捕获的异常（没有匹配的 catch 块的异常）在默认情况下将终止程序，意外异常（不与任何异常规范匹配的异常）也是如此。

RTTI（运行阶段类型信息）特性让程序能够检测对象的类型。dynamic_cast 运算符用于将派生类指针转换为基类指针，其主要用途是确保可以安全地调用虚函数。Typeid 运算符返回一个 type_info 对象。可以对两个 typeid 的返回值进行比较，以确定对象是否为特定的类型，而返回的 type_info 对象可用于获得关于对象的信息。

与通用转换机制相比，dynamic_cast、static_cast、const_cast 和 reinterpret_cast 提供了更安全、更明确的类型转换。

15.7　复习题

1. 下面建立友元的尝试有什么错误？

a.
```
class snap {
    friend clasp;
     ...
};
class clasp { ... };
```

b.
```
class cuff {
public:
      void snip(muff &) { ... }
      ...
};
class muff {
     friend void cuff::snip(muff &);
     ...
};
```

c.
```
class muff {
      friend void cuff::snip(muff &);
     ...
};
class cuff {
public:
      void snip(muff &) { ... }
      ...
};
```

2. 您知道了如何建立相互类友元的方法。能够创建一种更为严格的友情关系，即类 B 只有部分成员是类 A 的友元，而类 A 只有部分成员是类 B 的友元吗？请解释原因。

3. 下面的嵌套类声明中可能存在什么问题？

```
class Ribs
{
private:
    class Sauce
    {
          int soy;
          int sugar;
   public:
          Sauce(int s1, int s2) : soy(s1), sugar(s2) { }
   };
   ...
};
```

4. throw 和 return 之间的区别何在？

5. 假设有一个从异常基类派生来的异常类层次结构，则应按什么样的顺序放置 catch 块？

6. 对于本章定义的 Grand、Superb 和 Magnificent 类，假设 pg 为 Grand *指针，并将其中某个类的对象的地址赋给了它，而 ps 为 Superb *指针，则下面两个代码示例的行为有什么不同？

```
if (ps = dynamic_cast<Superb *>(pg))
    ps->say();  // sample #1
```

```
if (typeid(*pg) == typeid(Superb))
    (Superb *) pg)->say();  // sample #2
```

7. static_cast 运算符与 dynamic_cast 运算符有什么不同？

15.8 编程练习

1. 对 Tv 和 Remote 类做如下修改：

a. 让它们互为友元；

b. 在 Remote 类中添加一个状态变量成员，该成员描述遥控器是处于常规模式还是互动模式；

c. 在 Remote 中添加一个显示模式的方法；

d. 在 Tv 类中添加一个对 Remote 中新成员进行切换的方法，该方法应仅当 TV 处于打开状态时才能运行。

编写一个小程序来测试这些新特性。

2. 修改程序清单 15.11，使两种异常类型都是从头文件<stdexcept>提供的 logic_error 类派生出来的类。让每个 what( )方法都报告函数名和问题的性质。异常对象不用存储错误的参数值，而只需支持 what( )方法。

3. 这个练习与编程练习 2 相同，但异常类是从一个这样的基类派生而来的：它是从 logic_error 派生而来的，并存储两个参数值。异常类应该有一个这样的方法：报告这些值以及函数名。程序使用一个 catch 块来捕获基类异常，其中任何一种从该基类异常派生而来的异常都将导致循环结束。

4. 程序清单 15.16 在每个 try 后面都使用两个 catch 块，以确保 nbad_index 异常导致方法 label_val( ) 被调用。请修改该程序，在每个 try 块后面只使用一个 catch 块，并使用 RTTI 来确保合适时调用 label_val( )。

第 16 章　string 类和标准模板库

本章内容包括：

- 标准 C++ string 类。
- 模板 auto_ptr、unique_ptr 和 shared_ptr。
- 标准模板库（STL）。
- 容器类。
- 迭代器。
- 函数对象（functor）。
- STL 算法。
- 模板 initializer_list。

至此您熟悉了 C++可重用代码的目标，这样做的一个很大的回报是可以重用别人编写的代码，这正是类库的用武之地。有很多商业 C++类库，也有一些库是 C++程序包自带的。例如，曾使用过的头文件 ostream 支持的输入/输出类。本章介绍一些其他可重用代码，它们将给编程工作带来快乐。

本书前面介绍过 string 类，本章将更深入地讨论它；然后介绍“智能指针”模板类，它们让管理动态内存更容易；接下来介绍标准模板库（STL），它是一组用于处理各种容器对象的模板。STL 演示了一种编程模式——泛型编程；最后，本章将介绍 C++11 新增的模板 initializer_list，它让您能够将初始化列表语法用于 STL 对象。

16.1　string 类

很多应用程序都需要处理字符串。C 语言在 string.h（在 C++中为 cstring）中提供了一系列的字符串函数，很多早期的 C++实现为处理字符串提供了自己的类。第 4 章介绍了 ANSI/ISO C++ string 类，而第 12 章创建了一个不大的 String 类，以说明设计表示字符串的类的某些方面。

string 类是由头文件 string 支持的（注意，头文件 string.h 和 cstring 支持对 C-风格字符串进行操纵的 C 库字符串函数，但不支持 string 类）。要使用类，关键在于知道它的公有接口，而 string 类包含大量的方法，其中包括了若干构造函数，用于将字符串赋给变量、合并字符串、比较字符串和访问各个元素的重载运算符以及用于在字符串中查找字符和子字符串的工具等。简而言之，string 类包含的内容很多。

16.1.1　构造字符串

先来看 string 的构造函数。毕竟，对于类而言，最重要的内容之一是，有哪些方法可用于创建其对象。程序清单 16.1 使用了 string 的 7 个构造函数（用 ctor 标识，这是传统 C++中构造函数的缩写）。表 16.1 简要地描述了这些构造函数，它首先使用顺序简要描述了程序清单 16.1 使用的 7 个构造函数，然后列出了 C++11 新增的两个构造函数。使用构造函数时都进行了简化，即隐藏了这样一个事实：string 实际上是模板具体化 basic_string<char>的一个 typedef，同时省略了与内存管理相关的参数（这将在本章后面和附录 F

中讨论）。size_type 是一个依赖于实现的整型，是在头文件 string 中定义的。string 类将 string::npos 定义为字符串的最大长度，通常为 unsigned int 的最大值。另外，表格中使用缩写 NBTS（null-terminated string）来表示以空字符结束的字符串——传统的 C 字符串。

表 16.1　string 类的构造函数

构造函数	描述
string(const char * s)	将 string 对象初始化为 s 指向的 NBTS
string(size_type n, char c)	创建一个包含 n 个元素的 string 对象，其中每个元素都被初始化为字符 c
string(const string & str)	将一个 string 对象初始化为 string 对象 str（复制构造函数）
string( )	创建一个默认的 sting 对象，长度为 0（默认构造函数）
string(const char * s, size_type n)	将 string 对象初始化为 s 指向的 NBTS 的前 n 个字符，即使超过了 NBTS 结尾
template<class Iter> string(Iter begin, Iter end)	将 string 对象初始化为区间[begin, end)内的字符，其中 begin 和 end 的行为就像指针，用于指定位置，范围包括 begin 在内，但不包括 end
string(const string & str, string size_type pos = 0, size_type n = npos)	将一个 string 对象初始化为对象 str 中从位置 pos 开始到结尾的字符，或从位置 pos 开始的 n 个字符
string(string && str) noexcept	这是 C++11 新增的，它将一个 string 对象初始化为 string 对象 str，并可能修改 str（移动构造函数）
string(initializer_list<char> il)	这是 C++11 新增的，它将一个 string 对象初始化为初始化列表 il 中的字符

程序清单 16.1　str1.cpp

```
// str1.cpp -- introducing the string class
#include <iostream>
#include <string>
// using string constructors

int main()
{
    using namespace std;
    string one("Lottery Winner!");      // ctor #1
    cout << one << endl;                // overloaded <<
    string two(20, '$');                // ctor #2
    cout << two << endl;
    string three(one);                  // ctor #3
    cout << three << endl;
    one += " Oops!";                    // overloaded +=
    cout << one << endl;
    two = "Sorry! That was ";
    three[0] = 'P';
    string four;                        // ctor #4
    four = two + three;                 // overloaded +, =
    cout << four << endl;
    char alls[] = "All's well that ends well";
    string five(alls,20);               // ctor #5
    cout << five << "!\n";
    string six(alls+6, alls + 10);      // ctor #6
    cout << six  << ", ";
    string seven(&five[6], &five[10]); // ctor #6 again
    cout << seven << "...\n";
    string eight(four, 7, 16);          // ctor #7
```

```
    cout << eight << " in motion!" << endl;
    return 0;
}
```

程序清单 16.1 中程序还使用了重载+=运算符，它将一个字符串附加到另一个字符串的后面；重载的=运算符用于将一个字符串赋给另一个字符串；重载的<<运算符用于显示 string 对象；重载的[ ]运算符用于访问字符串中的各个字符。

下面是程序清单 16.1 中程序的输出：

```
Lottery Winner!
$$$$$$$$$$$$$$$$$$$$
Lottery Winner!
Lottery Winner! Oops!
Sorry! That was Pottery Winner!
All's well that ends!
well, well...
That was Pottery in motion!
```

1. 程序说明

程序清单 16.1 中的程序首先演示了可以将 string 对象初始化为常规的 C-风格字符串，然后使用重载的<<运算符来显示它：

```
string one("Lottery Winner!");     // ctor #1
cout << one << endl;               // overloaded <<
```

接下来的构造函数将 string 对象 two 初始化为由 20 个$字符组成的字符串：

```
string two(20, '$');               // ctor #2
```

复制构造函数将 string 对象 three 初始化为 string 对象 one：

```
string three(one);                 // ctor #3
```

重载的+=运算符将字符串“Oops!”附加到字符串 one 的后面：

```
one += " Oops!";                   // overloaded +=
```

这里是将一个 C-风格字符串附加到一个 string 对象的后面。但+=运算符被多次重载，以便能够附加 string 对象和单个字符：

```
one += two;    // append a string object (not in program)
one += '!';    // append a type char value (not in program)
```

同样，=运算符也被重载，以便可以将 string 对象、C-风格字符串或 char 值赋给 string 对象：

```
two = "Sorry! That was "; // assign a C-style string
two = one;                // assign a string object (not in program)
two = '?';                // assign a char value (not in program)
```

重载[ ]运算符（就像第 12 章的 String 示例那样）使得可以使用数组表示法来访问 string 对象中的各个字符：

```
three[0] = 'P';
```

默认构造函数创建一个以后可对其进行赋值的空字符串：

```
string four;                       // ctor #4
four = two + three;                // overloaded +, =
```

第 2 行使用重载的+运算符创建了一个临时 string 对象，然后使用重载的=运算符将它赋给对象 four。正如所预料的，+运算符将其两个操作数组合成一个 string 对象。该运算符被多次重载，以便第二个操作数可以是 string 对象、C-风格字符串或 char 值。

第 5 个构造函数将一个 C-风格字符串和一个整数作为参数，其中的整数参数表示要复制多少个字符：

```
char alls[] = "All's well that ends well";
string five(alls,20);              // ctor #5
```

从输出可知，这里只使用了前 20 个字符（“All's well that ends”）来初始化 five 对象。正如表 16.1 指出的，如果字符数超过了 C-风格字符串的长度，仍将复制请求数目的字符。所以在上面的例子中，如果用 40 代替 20，将导致 15 个无用字符被复制到 five 的结尾处（即构造函数将内存中位于字符串“All's well that ends well”后面的内容作为字符）。

第 6 个构造函数有一个模板参数：

```
template<class Iter> string(Iter begin, Iter end);
```

begin 和 end 将像指针那样，指向内存中两个位置（通常，begin 和 end 可以是迭代器——广泛用于 STL 中的广义化指针）。构造函数将使用 begin 和 end 指向的位置之间的值，对 string 对象进行初始化。[begin, end) 来自数学中，意味着包括 begin，但不包括 end 在内的区间。也就是说，end 指向被使用的最后一个值后面的一个位置。请看下面的语句：

```
string six(alls+6, alls + 10);    // ctor #6
```

由于数组名相当于指针，所以 alls + 6 和 alls +10 的类型都是 char *，因此使用模板时，将用类型 char * 替换 Iter。第一个参数指向数组 alls 中的第一个 w，第二个参数指向第一个 well 后面的空格。因此，six 将被初始化为字符串“well”。图 16.1 说明了该构造函数的工作原理。

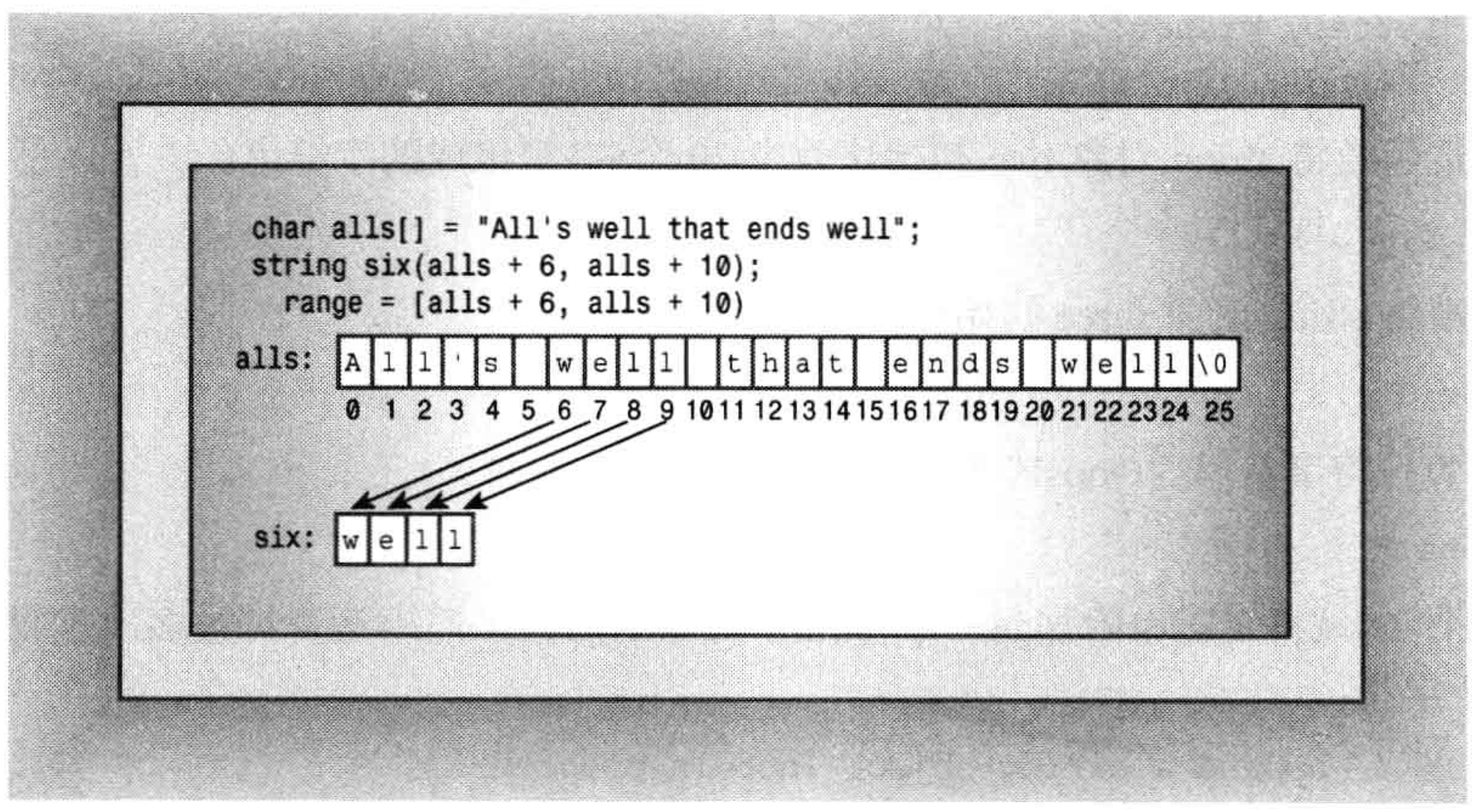

图 16.1 使用区间的 string 构造函数

现在假设要用这个构造函数将对象初始化为另一个 string 对象（假设为 five）的一部分内容，则下面的语句不管用：

```
string seven(five + 6, five + 10);
```

原因在于，对象名（不同于数组名）不会被看作是对象的地址，因此 five 不是指针，所以 five + 6 是没有意义的。然而，five[6]是一个 char 值，所以&five[6]是一个地址，因此可被用作该构造函数的一个参数。

```
string seven(&five[6], &five[10]);// ctor #6 again
```

第 7 个构造函数将一个 string 对象的部分内容复制到构造的对象中：

```
string eight(four, 7, 16);          // ctor #7
```

上述语句从 four 的第 8 个字符（位置 7）开始，将 16 个字符复制到 eight 中。

2. C++11 新增的构造函数

构造函数 string（string && str）类似于复制构造函数，导致新创建的 string 为 str 的副本。但与复制构

造函数不同的是，它不保证将 str 视为 const。这种构造函数被称为移动构造函数（move constructor）。在有些情况下，编译器可使用它而不是复制构造函数，以优化性能。第 18 章的“移动语义和右值引用”一节将讨论这个主题。

构造函数 string（initializer_list<char> il）让您能够将列表初始化语法用于 string 类。也就是说，它使得下面这样的声明是合法的：

```
string piano_man = {'L', 'i', 's','z','t'};
string comp_lang {'L', 'i', 's', 'p'};
```

就 string 类而言，这可能用处不大，因为使用 C-风格字符串更容易，但确实实现了让列表初始化语法普遍实用的意图。本章后面将更深入地讨论模板 initializer_list。

16.1.2　string 类输入

对于类，很有帮助的另一点是，知道有哪些输入方式可用。对于 C-风格字符串，有 3 种方式：

```
char info[100];
cin >> info;              // read a word
cin.getline(info, 100);   // read a line, discard \n
cin.get(info, 100);       // read a line, leave \n in queue
```

对于 string 对象，有两种方式：

```
string stuff;
cin >> stuff;             // read a word
getline(cin, stuff);      // read a line, discard \n
```

两个版本的 getline( )都有一个可选参数，用于指定使用哪个字符来确定输入的边界：

```
cin.getline(info,100,':');    // read up to :, discard :
getline(stuff, ':');          // read up to :, discard :
```

在功能上，它们之间的主要区别在于，string 版本的 getline( )将自动调整目标 string 对象的大小，使之刚好能够存储输入的字符：

```
char fname[10];
string lname;
cin >> fname;       // could be a problem if input size > 9 characters
cin >> lname;       // can read a very, very long word
cin.getline(fname, 10);  // may truncate input
getline(cin, fname);     // no truncation
```

自动调整大小的功能让 string 版本的 getline( )不需要指定读取多少个字符的数值参数。

在设计方面的一个区别是，读取 C-风格字符串的函数是 istream 类的方法，而 string 版本是独立的函数。这就是对于 C-风格字符串输入，cin 是调用对象；而对于 string 对象输入，cin 是一个函数参数的原因。这种规则也适用于>>形式，如果使用函数形式来编写代码，这一点将显而易见：

```
cin.operator>>(fname);     // ostream class method
operator>>(cin, lname);    // regular function
```

下面更深入地探讨一下 string 输入函数。正如前面指出的，这两个函数都自动调整目标 string 的大小，使之与输入匹配。但也存在一些限制。第一个限制因素是 string 对象的最大允许长度，由常量 string::npos 指定。这通常是最大的 unsigned int 值，因此对于普通的交互式输入，这不会带来实际的限制；但如果您试图将整个文件的内容读取到单个 string 对象中，这可能成为限制因素。第二个限制因素是程序可以使用的内存量。

string 版本的 getline( )函数从输入中读取字符，并将其存储到目标 string 中，直到发生下列三种情况之一：

- 到达文件尾，在这种情况下，输入流的 eofbit 将被设置，这意味着方法 fail( )和 eof( )都将返回 true；

- 遇到分界字符（默认为\n），在这种情况下，将把分界字符从输入流中删除，但不存储它；
- 读取的字符数达到最大允许值（string::npos 和可供分配的内存字节数中较小的一个），在这种情况下，将设置输入流的 failbit，这意味着方法 fail( )将返回 true。

输入流对象有一个统计系统，用于跟踪流的错误状态。在这个系统中，检测到文件尾后将设置 eofbit 寄存器，检测到输入错误时将设置 failbit 寄存器，出现无法识别的故障（如硬盘故障）时将设置 badbit 寄存器，一切顺利时将设置 goodbit 寄存器。第 17 章将更深入地讨论这一点。

string 版本的 operator>>( )函数的行为与此类似，只是它不断读取，直到遇到空白字符并将其留在输入队列中，而不是不断读取，直到遇到分界字符并将其丢弃。空白字符指的是空格、换行符和制表符，更普遍地说，是任何将其作为参数来调用 isspace( )时，该函数返回 ture 的字符。

本书前面有多个控制台 string 输入示例。由于用于 string 对象的输入函数使用输入流，能够识别文件尾，因此也可以使用它们来从文件中读取输入。程序清单 16.2 是一个从文件中读取字符串的简短示例，它假设文件中包含用冒号字符分隔的字符串，并使用指定分界符的 getline( )方法。然后，显示字符串并给它们编号，每个字符串占一行。

程序清单 16.2　strfile.cpp

```
// strfile.cpp -- read strings from a file
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
int main()
{
     using namespace std;
     ifstream fin;
     fin.open("tobuy.txt");
     if (fin.is_open() == false)
     {
        cerr << "Can't open file. Bye.\n";
        exit(EXIT_FAILURE);
     }
     string item;
     int count = 0;
     getline(fin, item, ':');
     while (fin)  // while input is good
     {
        ++count;
        cout << count <<": " << item << endl;
        getline(fin, item,':');
     }
     cout << "Done\n";
     fin.close();
     return 0;
}
```

下面是文件 tobuy.txt 的内容：

```
sardines:chocolate ice cream:pop corn:leeks:
cottage cheese:olive oil:butter:tofu:
```

通常，对于程序要查找的文本文件，应将其放在可执行程序或项目文件所在的目录中；否则必须提供完整的路径名。在 Windows 系统中，C-风格字符串中的转义序列\\表示一个斜杠：

```
fin.open("C:\\CPP\\Progs\\tobuy.txt"); // file = C:\CPP\Progs\tobuy.txt
```

下面是程序清单 16.2 中程序的输出：

```
1: sardines
2: chocolate ice cream
3: pop corn
4: leeks
5:
cottage cheese
6: olive oil
7: butter
8: tofu
9:

Done
```

注意，将:指定为分界字符后，换行符将被视为常规字符。因此文件 tobuy.txt 中第一行末尾的换行符将成为包含“cottage cheese”的字符串中的第一个字符。同样，第二行末尾的换行符是第 9 个输入字符串中唯一的内容。

16.1.3　使用字符串

现在，您知道可以使用不同方式来创建 string 对象、显示 string 对象的内容、将数据读取和附加到 string 对象中、给 string 对象赋值以及将两个 string 对象连结起来。除此之外，还能做些什么呢？

可以比较字符串。String 类对全部 6 个关系运算符都进行了重载。如果在机器排列序列中，一个对象位于另一个对象的前面，则前者被视为小于后者。如果机器排列序列为 ASCII 码，则数字将小于大写字符，而大写字符小于小写字符。对于每个关系运算符，都以三种方式被重载，以便能够将 string 对象与另一个 string 对象、C-风格字符串进行比较，并能够将 C-风格字符串与 string 对象进行比较：

```
string snake1("cobra");
string snake2("coral");
char snake3[20] = "anaconda";
if (snake1 < snake 2)          // operator<(const string &, const string &)
    ...
if (snake1 == snake3)          // operator==(const string &, const char *)
    ...
if (snake3 != snake2)          // operator!=(const char *, const string &)
    ...
```

可以确定字符串的长度。size( )和 length( )成员函数都返回字符串中的字符数：

```
if (snake1.length() == snake2.size())
    cout << "Both strings have the same length.\n"
```

为什么这两个函数完成相同的任务呢？length( )成员来自较早版本的 string 类，而 size( )则是为提供 STL 兼容性而添加的。

可以以多种不同的方式在字符串中搜索给定的子字符串或字符。表 16.2 简要地描述了 find( )方法的 4 个版本。如前所述，string ::npos 是字符串可存储的最大字符数，通常是无符号 int 或无符号 long 的最大取值。

表 16.2　重载的 find( )方法

方 法 原 型	描　　述
size_type find(const string & str, size_type pos = 0)const	从字符串的 pos 位置开始，查找子字符串 str。如果找到，则返回该子字符串首次出现时其首字符的索引；否则，返回 string :: npos
size_type find(const char * s, size_type pos = 0)const	从字符串的 pos 位置开始，查找子字符串 s。如果找到，则返回该子字符串首次出现时其首字符的索引；否则，返回 string :: npos

续表

方 法 原 型	描　　述
size_type find(const char * s, size_type pos = 0, size_type n)	从字符串的 pos 位置开始，查找 s 的前 n 个字符组成的子字符串。如果找到，则返回该子字符串首次出现时其首字符的索引；否则，返回 string :: npos
size_type find(char ch, size_type pos = 0)const	从字符串的 pos 位置开始，查找字符 ch。如果找到，则返回该字符首次出现的位置；否则，返回 string :: npos

string 库还提供了相关的方法：rfind( )、find_first_of( )、find_last_of( )、find_first_not_of( )和 find_last_not_of( )，它们的重载函数特征标都与 find( )方法相同。rfind( )方法查找子字符串或字符最后一次出现的位置；find_first_of( )方法在字符串中查找参数中任何一个字符首次出现的位置。例如，下面的语句返回 r 在“cobra”中的位置（即索引 3），因为这是“hark”中各个字母在“cobra”首次出现的位置：

```
int where = snake1.find_first_of("hark");
```

find_last_of( )方法的功能与此相同，只是它查找的是最后一次出现的位置。因此，下面的语句返回 a 在“cobra”中的位置：

```
int where = snake1.last_first_of("hark");
```

find_first_not_of( )方法在字符串中查找第一个不包含在参数中的字符，因此下面的语句返回 c 在“cobra”中的位置，因为“hark”中没有 c：

```
int where = snake1.find_first_not_of("hark");
```

在本章最后的练习中，您将了解 find_last_not_of( )。

还有很多其他的方法，这些方法足以创建一个非图形版本的 Hangman 拼字游戏。该游戏将一系列的单词存储在一个 string 对象数组中，然后随机选择一个单词，让人猜测单词的字母。如果猜错 6 次，玩家就输了。该程序使用 find( )函数来检查玩家的猜测，使用+=运算符创建一个 string 对象来记录玩家的错误猜测。为记录玩家猜对的情况，程序创建了一个单词，其长度与被猜的单词相同，但包含的是连字符。玩家猜对字符时，将用该字符替换相应的连字符。程序清单 16.3 列出了该程序的代码。

程序清单 16.3　hangman.cpp

```
// hangman.cpp -- some string methods
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
#include <cctype>
using std::string;
const int NUM = 26;
const string wordlist[NUM] = {"apiary", "beetle", "cereal",
    "danger", "ensign", "florid", "garage", "health", "insult",
    "jackal", "keeper", "loaner", "manage", "nonce", "onset",
    "plaid", "quilt", "remote", "stolid", "train", "useful",
    "valid", "whence", "xenon", "yearn", "zippy"};

int main()
{
    using std::cout;
    using std::cin;
    using std::tolower;
    using std::endl;
    std::srand(std::time(0));
    char play;
    cout << "Will you play a word game? <y/n> ";
```

```
    cin >> play;
    play = tolower(play);
    while (play == 'y')
    {
        string target = wordlist[std::rand() % NUM];
        int length = target.length();
        string attempt(length, '-');
        string badchars;
        int guesses = 6;
        cout << "Guess my secret word. It has " << length
                << " letters, and you guess\n"
                << "one letter at a time. You get " << guesses
        << " wrong guesses.\n";
    cout << "Your word: " << attempt << endl;
    while (guesses > 0 && attempt != target)
    {
        char letter;
        cout << "Guess a letter: ";
        cin >> letter;
        if (badchars.find(letter) != string::npos
            || attempt.find(letter) != string::npos)
        {
            cout << "You already guessed that. Try again.\n";
                continue;
        }
        int loc = target.find(letter);
        if (loc == string::npos)
        {
            cout << "Oh, bad guess!\n";
            --guesses;
            badchars += letter; // add to string
        }
        else
        {
            cout << "Good guess!\n";
            attempt[loc]=letter;
            // check if letter appears again
            loc = target.find(letter, loc + 1);
            while (loc != string::npos)
            {
                attempt[loc]=letter;
                loc = target.find(letter, loc + 1);
            }
        }
        cout << "Your word: " << attempt << endl;
        if (attempt != target)
        {
            if (badchars.length() > 0)
                cout << "Bad choices: " << badchars << endl;
            cout << guesses << " bad guesses left\n";
        }
    }
    if (guesses > 0)
        cout << "That's right!\n";
    else
```

```
    cout << "Sorry, the word is " << target << ".\n";
        cout << "Will you play another? <y/n> ";
        cin >> play;
        play = tolower(play);
    }

    cout << "Bye\n";

    return 0;
}
```

程序清单 16.3 中程序的运行情况如下：

```
Will you play a word game? <y/n> y
Guess my secret word. It has 6 letters, and you guess
one letter at a time. You get 6 wrong guesses.
Your word: ------
Guess a letter: e
Oh, bad guess!
Your word: ------
Bad choices: e
5 bad guesses left
Guess a letter: a
Good guess!
Your word: a--a--
Bad choices: e
5 bad guesses left
Guess a letter: t
Oh, bad guess!
Your word: a--a--
Bad choices: et
4 bad guesses left
Guess a letter: r
Good guess!
Your word: a--ar-
Bad choices: et
4 bad guesses left
Guess a letter: y
Good guess!
Your word: a--ary
Bad choices: et
4 bad guesses left
Guess a letter: i
Good guess!
Your word: a-iary
Bad choices: et
4 bad guesses left
Guess a letter: p
Good guess!
Your word: apiary
That's right!
Will you play another? <y/n> n
Bye
```

程序说明

在程序清单 16.3 中，由于关系运算符被重载，因此可以像对待数值变量那样对待字符串：

```
while (guesses > 0 && attempt != target)
```

与对 C-风格字符串使用 strcmp( )相比，这样简单些。

该程序使用 find( )来检查玩家以前是否猜过某个字符。如果是，则它要么位于 badchars 字符串（猜错）中，要么位于 attempt 字符串（猜对）中：

```
if (badchars.find(letter) != string::npos
    || attempt.find(letter) != string::npos)
```

npos 变量是 string 类的静态成员，它的值是 string 对象能存储的最大字符数。由于索引从 0 开始，所以它比最大的索引值大 1，因此可以使用它来表示没有查找到字符或字符串。

该程序利用了这样一个事实：+=运算符的某个重载版本使得能够将一个字符附加到字符串中：

```
badchars += letter;  // append a char to a string object
```

该程序的核心是从检查玩家选择的字符是否位于被猜测的单词中开始的：

```
int loc = target.find(letter);
```

如果 loc 是一个有效的值，则可以将该字母放置在答案字符串的相应位置：

```
attempt[loc]=letter;
```

然而，由于字母在被猜测的单词中可能出现多次，所以程序必须一直进行检查。该程序使用了 find( )的第二个可选参数，该参数可以指定从字符串什么位置开始搜索。因为字母是在位置 loc 找到的，所以下一次搜索应从 loc+1 开始。while 循环使搜索一直进行下去，直到找不到该字符为止。如果 loc 位于字符串尾，则表明 find( )没有找到该字符。

```
// check if letter appears again
loc = target.find(letter, loc + 1);
while (loc != string::npos)
{
    attempt[loc]=letter;
    loc = target.find(letter, loc + 1);
}
```

16.1.4　string 还提供了哪些功能

string 库提供了很多其他的工具，包括完成下述功能的函数：删除字符串的部分或全部内容、用一个字符串的部分或全部内容替换另一个字符串的部分或全部内容、将数据插入到字符串中或删除字符串中的数据、将一个字符串的部分或全部内容与另一个字符串的部分或全部内容进行比较、从字符串中提取子字符串、将一个字符串中的内容复制到另一个字符串中、交换两个字符串的内容。这些函数中的大多数都被重载，以便能够同时处理 C-风格字符串和 string 对象。附录 F 简要地介绍了 string 库中的函数。

首先来看自动调整大小的功能。在程序清单 16.3 中，每当程序将一个字母附加到字符串末尾时将发生什么呢？不能仅仅将已有的字符串加大，因为相邻的内存可能被占用了。因此，可能需要分配一个新的内存块，并将原来的内容复制到新的内存单元中。如果执行大量这样的操作，效率将非常低，因此很多 C++实现分配一个比实际字符串大的内存块，为字符串提供了增大空间。然而，如果字符串不断增大，超过了内存块的大小，程序将分配一个大小为原来两倍的新内存块，以提供足够的增大空间，避免不断地分配新的内存块。方法 capacity( )返回当前分配给字符串的内存块的大小，而 reserve( )方法让您能够请求内存块的最小长度。程序清单 16.4 是一个使用这些方法的示例。

程序清单 16.4　str2.cpp

```
// str2.cpp -- capacity() and reserve()
#include <iostream>
#include <string>
```

```
int main()
{
    using namespace std;
    string empty;
    string small = "bit";
    string larger = "Elephants are a girl's best friend";
    cout << "Sizes:\n";
    cout << "\tempty: " << empty.size() << endl;
    cout << "\tsmall: " << small.size() << endl;
    cout << "\tlarger: " << larger.size() << endl;
    cout << "Capacities:\n";
    cout << "\tempty: " << empty.capacity() << endl;
    cout << "\tsmall: " << small.capacity() << endl;
    cout << "\tlarger: " << larger.capacity() << endl;
    empty.reserve(50);
    cout << "Capacity after empty.reserve(50): "
         << empty.capacity() << endl;
    return 0;
}
```

下面是使用某种 C++实现时，程序清单 16.4 中程序的输出：

```
Sizes:
        empty: 0
        small: 3
        larger: 34
Capacities:
        empty: 15
        small: 15
        larger: 47
Capacity after empty.reserve(50): 63
```

注意，该实现使用的最小容量为 15 个字符，这比标准容量选择（16 的倍数）小 1。其他实现可能做出不同的选择。

如果您有 string 对象，但需要 C-风格字符串，该如何办呢？例如，您可能想打开一个其名称存储在 string 对象中的文件：

```
string filename;
cout << "Enter file name: ";
cin >> filename;
ofstream fout;
```

不幸的是，open( )方法要求使用一个 C-风格字符串作为参数；幸运的是，c_str( )方法返回一个指向 C-风格字符串的指针，该 C-风格字符串的内容与用于调用 c_str( )方法的 string 对象相同。因此可以这样做：

```
fout.open(filename.c_str());
```

16.1.5 字符串种类

本节将 string 类看作是基于 char 类型的。事实上，正如前面指出的，string 库实际上是基于一个模板类的：

```
template<class charT, class traits = char _traits<charT>,
        class Allocator = allocator<charT> >
basic_string {...};
```

模板 basic_string 有 4 个具体化，每个具体化都有一个 typedef 名称：

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
typedef basic_string<char16_t> u16string;   // C++11
typedef basic_string<char32_t> u32string ;  // C++11
```

这让您能够使用基于类型 wchar_t、char16_t、char32_t 和 char 的字符串。甚至可以开发某种类似字符的类，并对它使用 basic_string 类模板（只要它满足某些要求）。traits 类描述关于选定字符类型的特定情况，如如何对值进行比较。对于 wchar_t、char16_t、char32_t 和 char 类型，有预定义的 char_traits 模板具体化，它们都是 traits 的默认值。Allocator 是一个管理内存分配的类。对于各种字符类型，都有预定义的 allocator 模板具体化，它们都是默认的。它们使用 new 和 delete。

16.2　智能指针模板类

智能指针是行为类似于指针的类对象，但这种对象还有其他功能。本节介绍三个可帮助管理动态内存分配的智能指针模板。先来看需要哪些功能以及这些功能是如何实现的。请看下面的函数：

```
void remodel(std::string & str)
{
    std::string * ps = new std::string(str);
    ...
    str = ps;
    return;
}
```

您可能发现了其中的缺陷。每当调用时，该函数都分配堆中的内存，但从不收回，从而导致内存泄漏。您可能也知道解决之道——只要别忘了在 return 语句前添加下面的语句，以释放分配的内存即可：

```
delete ps;
```

然而，但凡涉及“别忘了”的解决方法，很少是最佳的。因为您有时可能忘了，有时可能记住了，但可能在不经意间删除或注释掉了这些代码。即使确实没有忘记，也可能有问题。请看下面的变体：

```
void remodel(std::string & str)
{
    std::string * ps = new std::string(str);
    ...
    if (weird_thing())
        throw exception();
    str = *ps;
    delete ps;
    return;
}
```

当出现异常时，delete 将不被执行，因此也将导致内存泄漏。

可以按第 14 章介绍的方式修复这种问题，但如果有更灵巧的解决方法就好了。来看一下需要些什么。当 remodel( )这样的函数终止（不管是正常终止，还是由于出现了异常而终止），本地变量都将从栈内存中删除——因此指针 ps 占据的内存将被释放。如果 ps 指向的内存也被释放，那该有多好啊。如果 ps 有一个析构函数，该析构函数将在 ps 过期时释放它指向的内存。因此，ps 的问题在于，它只是一个常规指针，不是有析构函数的类对象。如果它是对象，则可以在对象过期时，让它的析构函数删除指向的内存。这正是 auto_ptr、unique_ptr 和 shared_ptr 背后的思想。模板 auto_ptr 是 C++98 提供的解决方案，C++11 已将其摒弃，并提供了另外两种解决方案。然而，虽然 auto_ptr 被摒弃，但它已使用了多年；同时，如果您的编译器不支持其他两种解决方案，auto_ptr 将是唯一的选择。

16.2.1 使用智能指针

这三个智能指针模板（auto_ptr、unique_ptr 和 shared_ptr）都定义了类似指针的对象，可以将 new 获得（直接或间接）的地址赋给这种对象。当智能指针过期时，其析构函数将使用 delete 来释放内存。因此，如果将 new 返回的地址赋给这些对象，将无需记住稍后释放这些内存：在智能指针过期时，这些内存将自动被释放。图 16.2 说明了 auto_ptr 和常规指针在行为方面的差别；share_ptr 和 unique_ptr 的行为与 auto_ptr 相同。

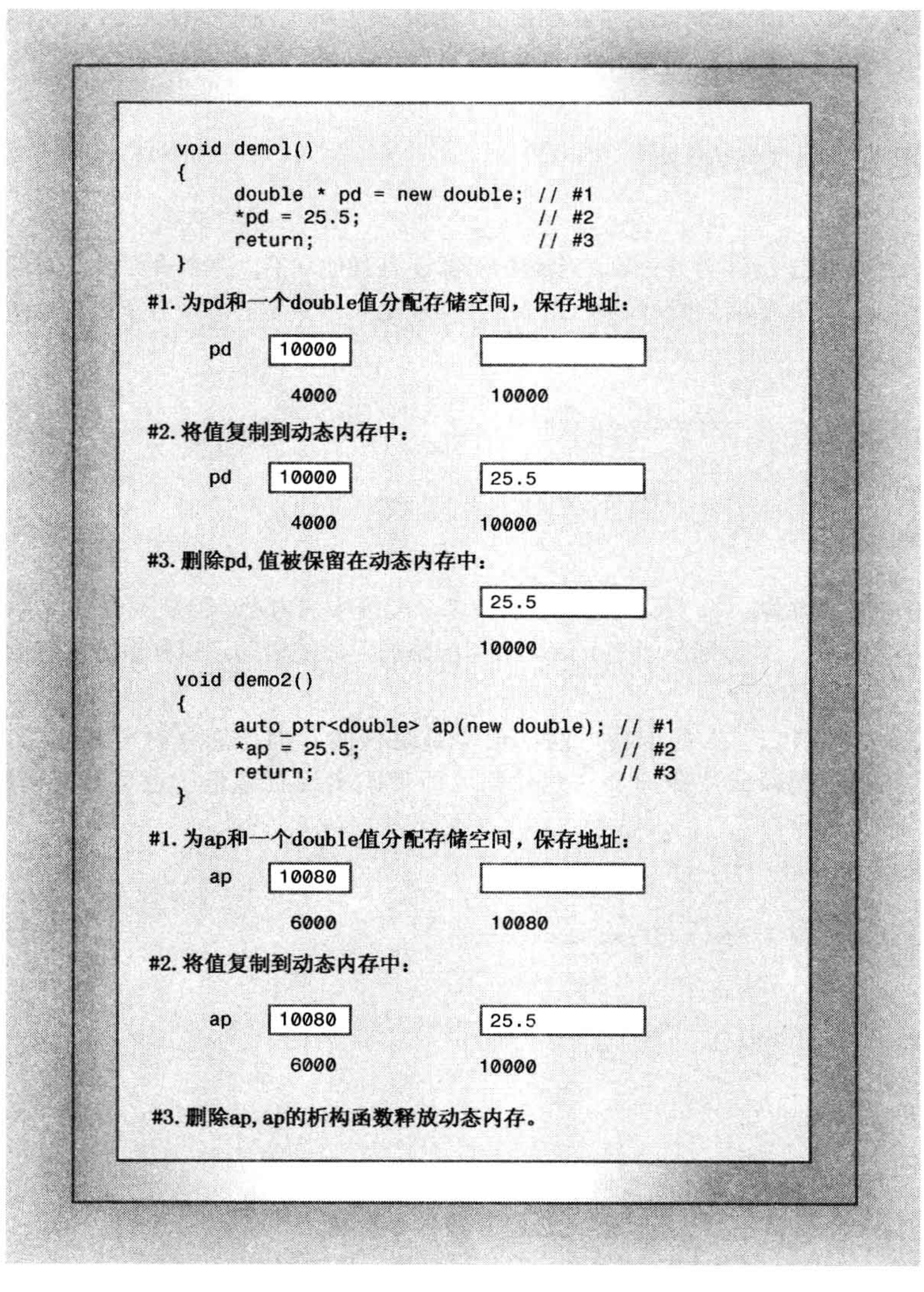

图 16.2 常规指针与 auto_ptr

要创建智能指针对象，必须包含头文件 memory，该文件模板定义。然后使用通常的模板语法来实例化所需类型的指针。例如，模板 auto_ptr 包含如下构造函数：

```
template<class X> class auto_ptr {
public:
    explicit auto_ptr(X* p =0) throw();
...};
```

本书前面说过，throw( )意味着构造函数不会引发异常；与 auto_ptr 一样，throw()也被摒弃。因此，请求 X 类型的 auto_ptr 将获得一个指向 X 类型的 auto_ptr：

```
auto_ptr<double> pd(new double);  // pd an auto_ptr to double
                                  // (use in place of double * pd)
auto_ptr<string> ps(new string);  // ps an auto_ptr to string
                                  // (use in place of string * ps)
```

new double 是 new 返回的指针，指向新分配的内存块。它是构造函数 auto_ptr<double>的参数，即对应于原型中形参 p 的实参。同样，new string 也是构造函数的实参。其他两种智能指针使用同样的语法：

```
unique_ptr<double> pdu(new double);  // pdu an unique_ptr to double
shared_ptr<string> pss(new string);  // pss a shared_ptr to string
```

因此，要转换 remodel( )函数，应按下面 3 个步骤进行：

1．包含头文件 memory；

2．将指向 string 的指针替换为指向 string 的智能指针对象；

3．删除 delete 语句。

下面是使用 auto_ptr 修改该函数的结果：

```
#include <memory>
void remodel(std::string & str)
{
    std::auto_ptr<std::string> ps (new std::string(str));
    ...
    if (weird_thing())
        throw exception();
    str = *ps;
    // delete ps;  NO LONGER NEEDED
    return;
}
```

注意到智能指针模板位于名称空间 std 中。程序清单 16.5 是一个简单的程序，演示了如何使用全部三种智能指针。要编译该程序，您的编译器必须支持 C++11 新增的类 share_ptr 和 unique_ptr。每个智能指针都放在一个代码块内，这样离开代码块时，指针将过期。Report 类使用方法报告对象的创建和销毁。

程序清单 16.5　smrtptrs.cpp

```
// smrtptrs.cpp -- using three kinds of smart pointers
// requires support of C++11 shared_ptr and unique_ptr
#include <iostream>
#include <string>
#include <memory>

class Report
{
private:
    std::string str;
public:
    Report(const std::string s) : str(s)
          { std::cout << "Object created!\n"; }
    ~Report() { std::cout << "Object deleted!\n"; }
    void comment() const { std::cout << str << "\n"; }
};

int main()
{
```

```
    {
        std::auto_ptr<Report> ps (new Report("using auto_ptr"));
        ps->comment();   // use -> to invoke a member function
    }
    {
        std::shared_ptr<Report> ps (new Report("using shared_ptr"));
        ps->comment();
    }
    {
        std::unique_ptr<Report> ps (new Report("using unique_ptr"));
        ps->comment();
    }
    return 0;
}
```

该程序的输出如下：

```
Object created!
using auto_ptr
Object deleted!
Object created!
using shared_ptr
Object deleted!
Object created!
using unique_ptr
Object deleted!
```

所有智能指针类都一个 explicit 构造函数，该构造函数将指针作为参数。因此不需要自动将指针转换为智能指针对象：

```
shared_ptr<double> pd;
double *p_reg = new double;
pd = p_reg;                          // not allowed (implicit conversion)
pd = shared_ptr<double>(p_reg);      // allowed (explicit conversion
shared_ptr<double> pshared = p_reg; // not allowed (implicit conversion)
shared_ptr<double> pshared(p_reg);  // allowed (explicit conversion)
```

由于智能指针模板类的定义方式，智能指针对象的很多方面都类似于常规指针。例如，如果 ps 是一个智能指针对象，则可以对它执行解除引用操作（* ps）、用它来访问结构成员（ps->puffIndex）、将它赋给指向相同类型的常规指针。还可以将智能指针对象赋给另一个同类型的智能指针对象，但将引起一个问题，这将在下一节进行讨论。

但在此之前，先说说对全部三种智能指针都应避免的一点：

```
string vacation("I wandered lonely as a cloud.");
shared_ptr<string> pvac(&vacation);  // NO!
```

pvac 过期时，程序将把 delete 运算符用于非堆内存，这是错误的。

就程序清单 16.5 演示的情况而言，三种智能指针都能满足要求，但情况并非总是这样简单。

16.2.2 有关智能指针的注意事项

为何有三种智能指针呢？实际上有 4 种，但本书不讨论 weak_ptr。为何摒弃 auto_ptr 呢？

先来看下面的赋值语句：

```
auto_ptr<string> ps (new string("I reigned lonely as a cloud."));
auto_ptr<string> vocation;
vocation = ps;
```

上述赋值语句将完成什么工作呢？如果 ps 和 vocation 是常规指针，则两个指针将指向同一个 string 对

象。这是不能接受的，因为程序将试图删除同一个对象两次——一次是 ps 过期时，另一次是 vocation 过期时。要避免这种问题，方法有多种。

- 定义赋值运算符，使之执行深复制。这样两个指针将指向不同的对象，其中的一个对象是另一个对象的副本。
- 建立所有权（ownership）概念，对于特定的对象，只能有一个智能指针可拥有它，这样只有拥有对象的智能指针的构造函数会删除该对象。然后，让赋值操作转让所有权。这就是用于 auto_ptr 和 unique_ptr 的策略，但 unique_ptr 的策略更严格。
- 创建智能更高的指针，跟踪引用特定对象的智能指针数。这称为引用计数（reference counting）。例如，赋值时，计数将加 1，而指针过期时，计数将减 1。仅当最后一个指针过期时，才调用 delete。这是 shared_ptr 采用的策略。

当然，同样的策略也适用于复制构造函数。

每种方法都有其用途。程序清单 16.6 是一个不适合使用 auto_ptr 的示例。

程序清单 16.6　fowl.cpp

```
// fowl.cpp  -- auto_ptr a poor choice
#include <iostream>
#include <string>
#include <memory>

int main()
{
    using namespace std;
    auto_ptr<string> films[5] =
    {
        auto_ptr<string> (new string("Fowl Balls")),
        auto_ptr<string> (new string("Duck Walks")),
        auto_ptr<string> (new string("Chicken Runs")),
        auto_ptr<string> (new string("Turkey Errors")),
        auto_ptr<string> (new string("Goose Eggs"))
    };
    auto_ptr<string> pwin;
    pwin = films[2];   // films[2] loses ownership

    cout << "The nominees for best avian baseball film are\n";
    for (int i = 0; i < 5; i++)
        cout << *films[i] << endl;
    cout << "The winner is " << *pwin << "!\n";
    cin.get();
    return 0;
}
```

下面是该程序的输出：

```
The nominees for best avian baseball film are
Fowl Balls
Duck Walks
Segmentation fault (core dumped)
```

消息 core dumped 表明，错误地使用 auto_ptr 可能导致问题（这种代码的行为是不确定的，其行为可能随系统而异）。这里的问题在于，下面的语句将所有权从 films[2]转让给 pwin：

```
pwin = films[2];   // films[2] loses ownership
```

这导致 films[2]不再引用该字符串。在 auto_ptr 放弃对象的所有权后，便可能使用它来访问该对象。当程序打印 films[2]指向的字符串时，却发现这是一个空指针，这显然讨厌的意外。

如果在程序清单 16.6 中使用 shared_ptr 代替 auto_ptr（这要求编译器支持 C++11 新增的 shared_ptr 类），则程序将正常运行，其输出如下：

```
The nominees for best avian baseball film are
Fowl Balls
Duck Walks
Chicken Runs
Turkey Errors
Goose Eggs
The winner is Chicken Runs!
```

差别在于程序的如下部分：

```
shared_ptr<string> pwin;
pwin = films[2];
```

这次 pwin 和 films[2]指向同一个对象，而引用计数从 1 增加到 2。在程序末尾，后声明的 pwin 首先调用其析构函数，该析构函数将引用计数降低到 1。然后，shared_ptr 数组的成员被释放，对 filmsp[2]调用析构函数时，将引用计数降低到 0，并释放以前分配的空间。

因此使用 shared_ptr 时，程序清单 16.6 运行正常；而使用 auto_ptr 时，该程序在运行阶段崩溃。如果使用 unique_ptr，结果将如何呢？与 auto_ptr 一样，unique_ptr 也采用所有权模型。但使用 unique_ptr 时，程序不会等到运行阶段崩溃，而在编译器因下述代码行出现错误：

```
pwin = films[2];
```

显然，该进一步探索 auto_ptr 和 unique_ptr 之间的差别。

16.2.3 unique_ptr 为何优于 auto_ptr

请看下面的语句：

```
auto_ptr<string> p1(new string("auto");  //#1
auto_ptr<string> p2;                     //#2
p2 = p1;                                 //#3
```

在语句#3 中，p2 接管 string 对象的所有权后，p1 的所有权将被剥夺。前面说过，这是件好事，可防止 p1 和 p2 的析构函数试图删除同一个对象；但如果程序随后试图使用 p1，这将是件坏事，因为 p1 不再指向有效的数据。

下面来看使用 unique_ptr 的情况：

```
unique_ptr<string> p3(new string("auto");  //#4
unique_ptr<string> p4;                     //#5
p4 = p3;                                   //#6
```

编译器认为语句#6 非法，避免了 p3 不再指向有效数据的问题。因此，unique_ptr 比 auto_ptr 更安全（编译阶段错误比潜在的程序崩溃更安全）。

但有时候，将一个智能指针赋给另一个并不会留下危险的悬挂指针。假设有如下函数定义：

```
unique_ptr<string> demo(const char * s)
{
    unique_ptr<string> temp(new string(s));
    return temp;
}
```

并假设编写了如下代码：

```
unique_ptr<string> ps;
ps = demo("Uniquely special");
```

demo( )返回一个临时 unique_ptr，然后 ps 接管了原本归返回的 unique_ptr 所有的对象，而返回的 unique_ptr 被销毁。这没有问题，因为 ps 拥有了 string 对象的所有权。但这里的另一个好处是，demo( )返回的临时 unique_ptr 很快被销毁，没有机会使用它来访问无效的数据。换句话说，没有理由禁止这种赋值。神奇的是，编译器确实允许这种赋值！

总之，程序试图将一个 unique_ptr 赋给另一个时，如果源 unique_ptr 是个临时右值，编译器允许这样做；如果源 unique_ptr 将存在一段时间，编译器将禁止这样做：

```
using namespace std;
unique_ptr< string> pu1(new string "Hi ho!");
unique_ptr< string> pu2;
pu2 = pu1;                                  //#1 not allowed
unique_ptr<string> pu3;
pu3 = unique_ptr<string>(new string "Yo!");  //#2 allowed
```

语句#1 将留下悬挂的 unique_ptr（pu1），这可能导致危害。语句#2 不会留下悬挂的 unique_ptr，因为它调用 unique_ptr 的构造函数，该构造函数创建的临时对象在其所有权转让给 pu 后就会被销毁。这种随情况而异的行为表明，unique_ptr 优于允许两种赋值的 auto_ptr。这也是禁止（只是一种建议，编译器并不禁止）在容器对象中使用 auto_ptr，但允许使用 unique_ptr 的原因。如果容器算法试图对包含 unique_ptr 的容器执行类似于语句#1 的操作，将导致编译错误；如果算法试图执行类似于语句#2 的操作，则不会有任何问题。而对于 auto_ptr，类似于语句#1 的操作可能导致不确定的行为和神秘的崩溃。

当然，您可能确实想执行类似于语句#1 的操作。仅当以非智能的方式使用遗弃的智能指针（如解除引用时），这种赋值才不安全。要安全地重用这种指针，可给它赋新值。C++有一个标准库函数 std::move( )，让您能够将一个 unique_ptr 赋给另一个。下面是一个使用前述 demo( )函数的例子，该函数返回一个 unique_ptr<string>对象：

```
using namespace std;
unique_ptr<string> ps1, ps2;
ps1 = demo("Uniquely special");
ps2 = move(ps1);                    // enable assignment
ps1 = demo(" and more");
cout << *ps2 << *ps1 << endl;
```

您可能会问，unique_ptr 如何能够区分安全和不安全的用法呢？答案是它使用了 C++11 新增的移动构造函数和右值引用，这将在第 18 章讨论。

相比于 auto_ptr，unique_ptr 还有另一个优点。它有一个可用于数组的变体。别忘了，必须将 delete 和 new 配对，将 delete []和 new [ ]配对。模板 auto_ptr 使用 delete 而不是 delete [ ]，因此只能与 new 一起使用，而不能与 new [ ]一起使用。但 unique_ptr 有使用 new [ ]和 delete [ ]的版本：

```
std::unique_ptr< double[]>pda(new double(5));  // will use delete []
```

警告：使用 new 分配内存时，才能使用 auto_ptr 和 shared_ptr，使用 new [ ]分配内存时，不能使用它们。不使用 new 分配内存时，不能使用 auto_ptr 或 shared_ptr；不使用 new 或 new []分配内存时，不能使用 unique_ptr。

16.2.4　选择智能指针

应使用哪种智能指针呢？如果程序要使用多个指向同一个对象的指针，应选择 shared_ptr。这样的情况包括：有一个指针数组，并使用一些辅助指针来标识特定的元素，如最大的元素和最小的元素；两个对象包含都指向第三个对象的指针；STL 容器包含指针。很多 STL 算法都支持复制和赋值操作，这些操作可用于 shared_ptr，但不能用于 unique_ptr（编译器发出警告）和 auto_ptr（行为不确定）。如果您的编译器没有提供 shared_ptr，可使用 Boost 库提供的 shared_ptr。

如果程序不需要多个指向同一个对象的指针，则可使用 unique_ptr。如果函数使用 new 分配内存，并

返回指向该内存的指针，将其返回类型声明为 unique_ptr 是不错的选择。这样，所有权将转让给接受返回值的 unique_ptr，而该智能指针将负责调用 delete。可将 unique_ptr 存储到 STL 容器中，只要不调用将一个 unique_ptr 复制或赋给另一个的方法或算法（如 sort( )）。例如，可在程序中使用类似于下面的代码段，这里假设程序包含正确的 include 和 using 语句：

```
unique_ptr<int> make_int(int n)
{
    return unique_ptr<int>(new int(n));
}
void show(unique_ptr<int> & pi)              // pass by reference
{
    cout << *a << ' ';
}
int main()
{
...
    vector<unique_ptr<int> > vp(size);
    for (int i = 0; i < vp.size(); i++)
        vp[i] = make_int(rand() % 1000);     // copy temporary unique_ptr
    vp.push_back(make_int(rand() % 1000))    // ok because arg is temporary
    for_each(vp.begin(), vp.end(), show);    // use for_each()
...
}
```

其中的 push_back( )调用没有问题，因为它返回一个临时 unique_ptr，该 unique_ptr 被赋给 vp 中的一个 unique_ptr。另外，如果按值而不是按引用给 show( )传递对象，for_each( )语句将非法，因为这将导致使用一个来自 vp 的非临时 unique_ptr 初始化 pi，而这是不允许的。前面说过，编译器将发现错误使用 unique_ptr 的企图。

在 unique_ptr 为右值时，可将其赋给 shared_ptr，这与将一个 unique_ptr 赋给另一个需要满足的条件相同。与前面一样，在下面的代码中，make_int( )的返回类型为 unique_ptr<int>：

```
unique_ptr<int> pup(make_int(rand() % 1000);  // ok
shared_ptr<int> spp(pup);                     // not allowed, pup an lvalue
shared_ptr<int> spr(make_int(rand() % 1000);  // ok
```

模板 shared_ptr 包含一个显式构造函数，可用于将右值 unique_ptr 转换为 shared_ptr。shared_ptr 将接管原来归 unique_ptr 所有的对象。

在满足 unique_ptr 要求的条件时，也可使用 auto_ptr，但 unique_ptr 是更好的选择。如果您的编译器没有提供 unique_ptr，可考虑使用 BOOST 库提供的 scoped_ptr，它与 unique_ptr 类似。

16.3 标准模板库

STL 提供了一组表示容器、迭代器、函数对象和算法的模板。容器是一个与数组类似的单元，可以存储若干个值。STL 容器是同质的，即存储的值的类型相同；算法是完成特定任务（如对数组进行排序或在链表中查找特定值）的处方；迭代器能够用来遍历容器的对象，与能够遍历数组的指针类似，是广义指针；函数对象是类似于函数的对象，可以是类对象或函数指针（包括函数名，因为函数名被用作指针）。STL 使得能够构造各种容器（包括数组、队列和链表）和执行各种操作（包括搜索、排序和随机排列）。

Alex Stepanov 和 Meng Lee 在 Hewlett-Packard 实验室开发了 STL，并于 1994 年发布其实现。ISO/ANSI C++委员会投票同意将其作为 C++标准的组成部分。STL 不是面向对象的编程，而是一种不同的编程模式——泛型编程（generic programming）。这使得 STL 在功能和方法方面都很有趣。关于 STL 的信息很

多，无法用一章的篇幅全部介绍，所以这里将介绍一些有代表性的例子，并领会泛型编程方法的精神。先来看几个具体的例子，让您对容器、迭代器和算法有一些感性的认识，然后再介绍底层的设计理念，并简要地介绍 STL。附录 G 对各种 STL 方法和函数进行了总结。

16.3.1　模板类 vector

第 4 章简要地介绍了 vector 类，下面更详细地介绍它。在计算中，矢量（vector）对应数组，而不是第 11 章介绍的数学矢量（在数学中，可以使用 N 个分量来表示 N 维数学矢量，因此从这方面讲，数学矢量类似一个 N 维数组。然而，数学矢量还有一些计算机矢量不具备的其他特征，如内乘积和外乘积）。计算矢量存储了一组可随机访问的值，即可以使用索引来直接访问矢量的第 10 个元素，而不必首先访问前面第 9 个元素。所以 vector 类提供了与第 14 章介绍的 valarray 和 ArrayTP 以及第 4 章介绍的 array 类似的操作，即可以创建 vector 对象，将一个 vector 对象赋给另一个对象，使用[ ]运算符来访问 vector 元素。要使类成为通用的，应将它设计为模板类，STL 正是这样做的——在头文件 vector（以前为 vector.h）中定义了一个 vector 模板。

要创建 vector 模板对象，可使用通常的<type>表示法来指出要使用的类型。另外，vector 模板使用动态内存分配，因此可以用初始化参数来指出需要多少矢量：

```
#include vector
using namespace std;
vector<int> ratings(5);        // a vector of 5 ints
int n;
cin >> n;
vector<double> scores(n);      // a vector of n doubles
```

由于运算符[ ]被重载，因此创建 vector 对象后，可以使用通常的数组表示法来访问各个元素：

```
ratings[0] = 9;
for (int i = 0; i < n; i++)
    cout << scores[i] << endl;
```

分配器

与 string 类相似，各种 STL 容器模板都接受一个可选的模板参数，该参数指定使用哪个分配器对象来管理内存。例如，vector 模板的开头与下面类似：

```
template <class T, class Allocator = allocator<T> >
    class vector {...
```

如果省略该模板参数的值，则容器模板将默认使用 allocator<T>类。这个类使用 new 和 delete。

程序清单 16.7 是一个要求不高的应用程序，它使用了这个类。该程序创建了两个 vector 对象——一个是 int 规范，另一个是 string 规范，它们都包含 5 个元素。

程序清单 16.7　vect1.cpp

```
// vect1.cpp -- introducing the vector template
#include <iostream>
#include <string>
#include <vector>

const int NUM = 5;
int main()
{
    using std::vector;
    using std::string;
    using std::cin;
    using std::cout;
```

```
    vector<int> ratings(NUM);
    vector<string> titles(NUM);
    cout << "You will do exactly as told. You will enter\n"
         << NUM << " book titles and your ratings (0-10).\n";
    int i;
    for (i = 0; i < NUM; i++)
    {
        cout << "Enter title #" << i + 1 << ": ";
        getline(cin,titles[i]);
        cout << "Enter your rating (0-10): ";
        cin >> ratings[i];
        cin.get();
    }
    cout << "Thank you. You entered the following:\n"
          << "Rating\tBook\n";
    for (i = 0; i < NUM; i++)
    {
        cout << ratings[i] << "\t" << titles[i] << endl;
    }

    return 0;
}
```

程序清单 16.7 中程序的运行情况如下：

```
You will do exactly as told. You will enter
5 book titles and your ratings (0-10).
Enter title #1: The Cat Who Knew C++
Enter your rating (0-10): 6
Enter title #2: Felonious Felines
Enter your rating (0-10): 4
Enter title #3: Warlords of Wonk
Enter your rating (0-10): 3
Enter title #4: Don't Touch That Metaphor
Enter your rating (0-10): 5
Enter title #5: Panic Oriented Programming
Enter your rating (0-10): 8
Thank you. You entered the following:
Rating  Book
6       The Cat Who Knew C++
4       Felonious Felines
3       Warlords of Wonk
5       Don't Touch That Metaphor
8       Panic Oriented Programming
```

该程序使用 vector 模板只是为方便创建动态分配的数组。下一节将介绍一个使用更多类方法的例子。

16.3.2 可对矢量执行的操作

除分配存储空间外，vector 模板还可以完成哪些任务呢？所有的 STL 容器都提供了一些基本方法，其中包括 size( )——返回容器中元素数目、swap( )——交换两个容器的内容、begin( )——返回一个指向容器中第一个元素的迭代器、end( )——返回一个表示超过容器尾的迭代器。

什么是迭代器？它是一个广义指针。事实上，它可以是指针，也可以是一个可对其执行类似指针的操作——如解除引用（如 operator*( )）和递增（如 operator++( )）——的对象。稍后将知道，通过将指针广

义化为迭代器，让 STL 能够为各种不同的容器类（包括那些简单指针无法处理的类）提供统一的接口。每个容器类都定义了一个合适的迭代器，该迭代器的类型是一个名为 iterator 的 typedef，其作用域为整个类。例如，要为 vector 的 double 类型规范声明一个迭代器，可以这样做：

```
vector<double>::iterator pd;  // pd an iterator
```

假设 scores 是一个 vector<double>对象：

```
vector<double> scores;
```

则可以使用迭代器 pd 执行这样的操作：

```
pd = scores.begin();  // have pd point to the first element
*pd = 22.3;           // dereference pd and assign value to first element
++pd;                 // make pd point to the next element
```

正如您看到的，迭代器的行为就像指针。顺便说一句，还有一个 C++11 自动类型推断很有用的地方。例如，可以不这样做：

```
vector<double>::iterator pd = scores.begin();
```

而这样做：

```
auto pd = scores.begin();   // C++11 automatic type deduction
```

回到前面的示例。什么是超过结尾（past-the-end）呢？它是一种迭代器，指向容器最后一个元素后面的那个元素。这与 C-风格字符串最后一个字符后面的空字符类似，只是空字符是一个值，而“超过结尾”是一个指向元素的指针（迭代器）。end( )成员函数标识超过结尾的位置。如果将迭代器设置为容器的第一个元素，并不断地递增，则最终它将到达容器结尾，从而遍历整个容器的内容。因此，如果 scores 和 pd 的定义与前面的示例中相同，则可以用下面的代码来显示容器的内容：

```
for (pd = scores.begin(); pd != scores.end(); pd++)
    cout << *pd << endl;;
```

所有容器都包含刚才讨论的那些方法。vector 模板类也包含一些只有某些 STL 容器才有的方法。push_back( )是一个方便的方法，它将元素添加到矢量末尾。这样做时，它将负责内存管理，增加矢量的长度，使之能够容纳新的成员。这意味着可以编写这样的代码：

```
vector<double> scores;  // create an empty vector
double temp;
while (cin >> temp && temp >= 0)
    scores.push_back(temp);
cout << "You entered " << scores.size() << " scores.\n";
```

每次循环都给 scores 对象增加一个元素。在编写或运行程序时，无需了解元素的数目。只要能够取得足够的内存，程序就可以根据需要增加 scores 的长度。

erase( )方法删除矢量中给定区间的元素。它接受两个迭代器参数，这些参数定义了要删除的区间。了解 STL 如何使用两个迭代器来定义区间至关重要。第一个迭代器指向区间的起始处，第二个迭代器位于区间终止处的后一个位置。例如，下述代码删除第一个和第二个元素，即删除 begin( )和 begin( )+1 指向的元素（由于 vector 提供了随机访问功能，因此 vector 类迭代器定义了诸如 begin( )+2 等操作）：

```
scores.erase(scores.begin(), scores.begin() + 2);
```

如果 it1 和 it2 是迭代器，则 STL 文档使用[p1, p2)来表示从 p1 到 p2（不包括 p2）的区间。因此，区间[begin( ), end( )]将包括集合的所有内容（参见图 16.3），而区间[p1, p1)为空。[ )表示法并不是 C++的组成部分，因此不能在代码中使用，而只能出现在文档中。

注意：区间[it1, it2)由迭代器 it1 和 it2 指定，其范围为 it1 到 it2（不包括 it2）。

insert( )方法的功能与 erase( )相反。它接受 3 个迭代器参数，第一个参数指定了新元素的插入位置，第二个和第三个迭代器参数定义了被插入区间，该区间通常是另一个容器对象的一部分。例如，下面的代码将矢量 new_v 中除第一个元素外的所有元素插入到 old_v 矢量的第一个元素前面：

```
vector<int> old_v;
vector<int> new_v;
...
old_v.insert(old_v.begin(), new_v.begin() + 1, new_v.end());
```

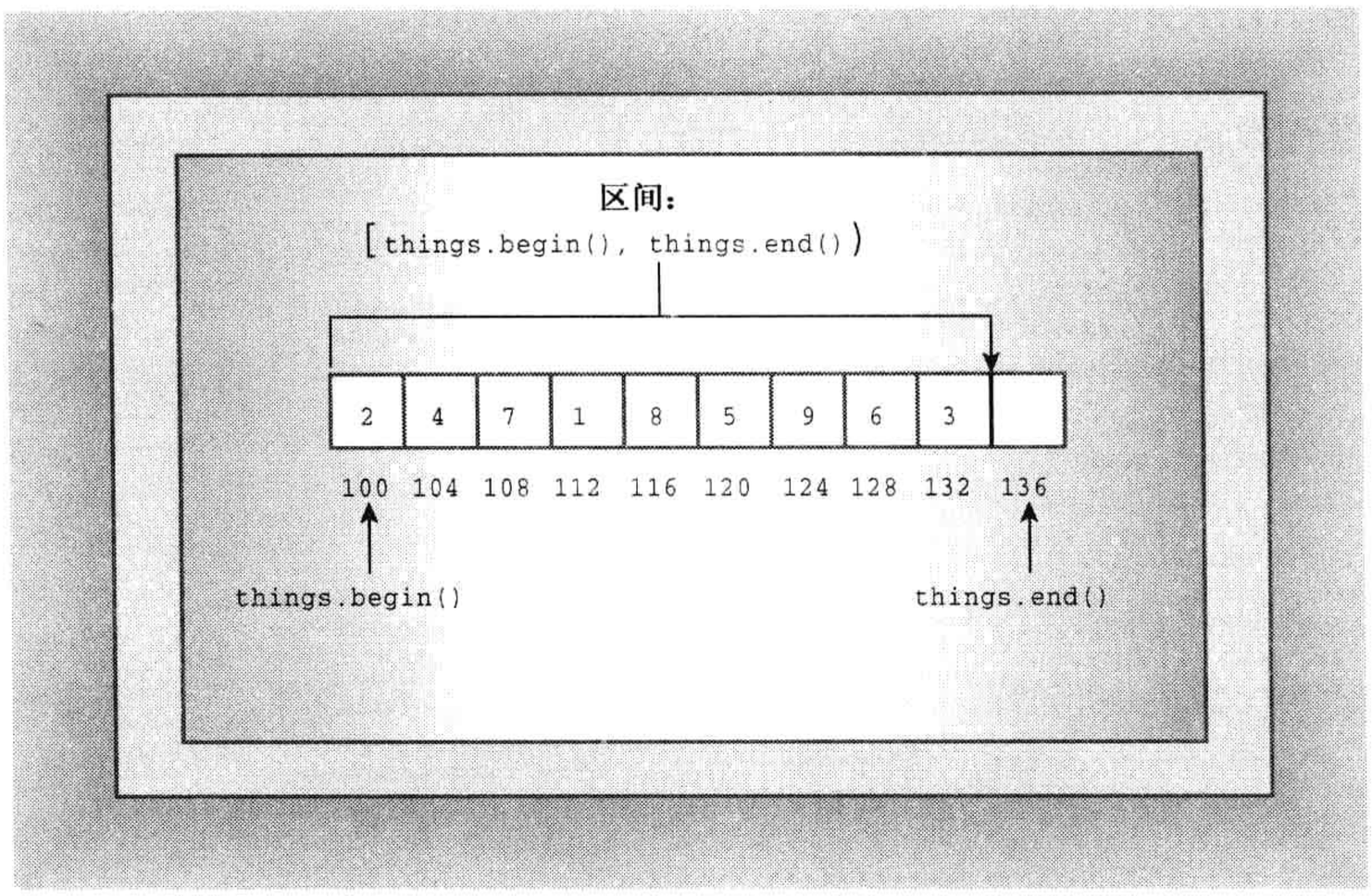

图 16.3　STL 的区间概念

顺便说一句，对于这种情况，拥有超尾元素是非常方便的，因为这使得在矢量尾部附加元素非常简单。下面的代码将新元素插入到 old.end( )前面，即矢量最后一个元素的后面。

```
old_v.insert(old_v.end(), new_v.begin() + 1, new_v.end());
```

程序清单 16.8 演示了 size( )、begin( )、end( )、push_back( )、erase( )和 insert( )的用法。为简化数据处理，将程序清单 16.7 中的 rating 和 title 组合成了一个 Review 结构，并使用 FillReview( )和 ShowReview( )函数来输入和输出 Review 对象。

程序清单 16.8　vect2.cpp

```
// vect2.cpp -- methods and iterators
#include <iostream>
#include <string>
#include <vector>

struct Review {
    std::string title;
    int rating;
};
bool FillReview(Review & rr);
void ShowReview(const Review & rr);

int main()
{
    using std::cout;
    using std::vector;
    vector<Review> books;
    Review temp;
    while (FillReview(temp))
```

```
        books.push_back(temp);
    int num = books.size();
    if (num > 0)
    {
        cout << "Thank you. You entered the following:\n"
            << "Rating\tBook\n";
        for (int i = 0; i < num; i++)
            ShowReview(books[i]);
        cout << "Reprising:\n"
            << "Rating\tBook\n";
        vector<Review>::iterator pr;
        for (pr = books.begin(); pr != books.end(); pr++)
            ShowReview(*pr);
        vector <Review> oldlist(books);     // copy constructor used
        if (num > 3)
        {
            // remove 2 items
            books.erase(books.begin() + 1, books.begin() + 3);
            cout << "After erasure:\n";
            for (pr = books.begin(); pr != books.end(); pr++)
                ShowReview(*pr);
            // insert 1 item
            books.insert(books.begin(), oldlist.begin() + 1,
                         oldlist.begin() + 2);
            cout << "After insertion:\n";
            for (pr = books.begin(); pr != books.end(); pr++)
                ShowReview(*pr);
            }
            books.swap(oldlist);
            cout << "Swapping oldlist with books:\n";
            for (pr = books.begin(); pr != books.end(); pr++)
                ShowReview(*pr);
        }
        else
            cout << "Nothing entered, nothing gained.\n";
        return 0;
    }

    bool FillReview(Review & rr)
    {
        std::cout << "Enter book title (quit to quit): ";
        std::getline(std::cin,rr.title);
        if (rr.title == "quit")
            return false;
        std::cout << "Enter book rating: ";
        std::cin >> rr.rating;
        if (!std::cin)
            return false;
        // get rid of rest of input line
        while (std::cin.get() != '\n')
            continue;
        return true;
    }

    void ShowReview(const Review & rr)
```

```
{
    std::cout << rr.rating << "\t" << rr.title << std::endl;
}
```

程序清单 16.8 中程序的运行情况如下：

```
Enter book title (quit to quit): The Cat Who Knew Vectors
Enter book rating: 5
Enter book title (quit to quit): Candid Canines
Enter book rating: 7
Enter book title (quit to quit): Warriors of Wonk
Enter book rating: 4
Enter book title (quit to quit): Quantum Manners
Enter book rating: 8
Enter book title (quit to quit): quit
Thank you. You entered the following:
Rating  Book
5       The Cat Who Knew Vectors
7       Candid Canines
4       Warriors of Wonk
8       Quantum Manners
Reprising:
Rating  Book
5       The Cat Who Knew Vectors
7       Candid Canines
4       Warriors of Wonk
8       Quantum Manners
After erasure:
5       The Cat Who Knew Vectors
8       Quantum Manners
After insertion:
7       Candid Canines
5       The Cat Who Knew Vectors
8       Quantum Manners
Swapping oldlist with books:
5       The Cat Who Knew Vectors
7       Candid Canines
4       Warriors of Wonk
8       Quantum Manners
```

16.3.3 对矢量可执行的其他操作

程序员通常要对数组执行很多操作，如搜索、排序、随机排序等。矢量模板类包含了执行这些常见的操作的方法吗？没有！STL 从更广泛的角度定义了非成员（non-member）函数来执行这些操作，即不是为每个容器类定义 find( )成员函数，而是定义了一个适用于所有容器类的非成员函数 find( )。这种设计理念省去了大量重复的工作。例如，假设有 8 个容器类，需要支持 10 种操作。如果每个类都有自己的成员函数，则需要定义 80（8*10）个成员函数。但采用 STL 方式时，只需要定义 10 个非成员函数即可。在定义新的容器类时，只要遵循正确的指导思想，则它也可以使用已有的 10 个非成员函数来执行查找、排序等操作。

另一方面，即使有执行相同任务的非成员函数，STL 有时也会定义一个成员函数。这是因为对有些操作来说，类特定算法的效率比通用算法高，因此，vector 的成员函数 swap( )的效率比非成员函数 swap( )高，但非成员函数让您能够交换两个类型不同的容器的内容。

下面来看 3 个具有代表性的 STL 函数：for_each( )、random_shuffle( )和 sort( )。for_each( )函数可用于很多容器类，它接受 3 个参数。前两个是定义容器中区间的迭代器，最后一个是指向函数的指针（更普遍

地说，最后一个参数是一个函数对象，函数对象将稍后介绍）。for_each( )函数将被指向的函数应用于容器区间中的各个元素。被指向的函数不能修改容器元素的值。可以用 for_each( )函数来代替 for 循环。例如，可以将代码：

```
vector<Review>::iterator pr;
for (pr = books.begin(); pr != books.end(); pr++)
    ShowReview(*pr);
```

替换为：

```
for_each(books.begin(), books.end(), ShowReview);
```

这样可避免显式地使用迭代器变量。

Random_shuffle( )函数接受两个指定区间的迭代器参数，并随机排列该区间中的元素。例如，下面的语句随机排列 books 矢量中所有元素：

```
random_shuffle(books.begin(), books.end());
```

与可用于任何容器类的 for_each 不同，该函数要求容器类允许随机访问，vector 类可以做到这一点。

sort( )函数也要求容器支持随机访问。该函数有两个版本，第一个版本接受两个定义区间的迭代器参数，并使用为存储在容器中的类型元素定义的<运算符，对区间中的元素进行操作。例如，下面的语句按升序对 coolstuff 的内容进行排序，排序时使用内置的<运算符对值进行比较：

```
vector<int> coolstuff;
...
sort(coolstuff.begin(), coolstuff.end());
```

如果容器元素是用户定义的对象，则要使用 sort( )，必须定义能够处理该类型对象的 operator<( )函数。例如，如果为 Review 提供了成员或非成员函数 operator<( )，则可以对包含 Review 对象的矢量进行排序。由于 Review 是一个结构，因此其成员是公有的，这样的非成员函数将为：

```
bool operator<(const Review & r1, const Review & r2)
{
    if (r1.title < r2.title)
        return true;
    else if (r1.title == r2.title && r1.rating < r2.rating)
        return true;
    else
        return false;
}
```

有了这样的函数后，就可以对包含 Review 对象（如 books）的矢量进行排序了：

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

上述版本的 operator<( )函数按 title 成员的字母顺序排序。如果 title 成员相同，则按照 rating 排序。然而，如果想按降序或是按 rating（而不是 title）排序，该如何办呢？可以使用另一种格式的 sort( )。它接受 3 个参数，前两个参数也是指定区间的迭代器，最后一个参数是指向要使用的函数的指针（函数对象），而不是用于比较的 operator<( )。返回值可转换为 bool，false 表示两个参数的顺序不正确。下面是一个例子：

```
bool WorseThan(const Review & r1, const Review & r2)
{
    if (r1.rating < r2.rating)
        return true;
    else
        return false;
}
```

有了这个函数后，就可以使用下面的语句将包含 Review 对象的 books 矢量按 rating 升序排列：

```
sort(books.begin(), books.end(), WorseThan);
```

注意，与 operator<( )相比，WorseThan( )函数执行的对 Review 对象进行排序的工作不那么完整。如果两个对象的 title 成员相同，operator<( )函数将按 rating 进行排序，而 WorseThan( )将它们视为相同。第一种排序称为全排序（total ordering），第二种排序称为完整弱排序（strict weak ordering）。在全排序中，如果 a<b 和 b<a 都不成立，则 a 和 b 必定相同。在完整弱排序中，情况就不是这样了。它们可能相同，也可能只是在某方面相同，如 WorseThan( )示例中的 rating 成员。所以在完整弱排序中，只能说它们等价，而不是相同。

程序清单 16.9 演示了这些 STL 函数的用法。

程序清单 16.9　vect3.cpp

```
// vect3.cpp -- using STL functions
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

struct Review {
    std::string title;
    int rating;
};

bool operator<(const Review & r1, const Review & r2);
bool worseThan(const Review & r1, const Review & r2);
bool FillReview(Review & rr);
void ShowReview(const Review & rr);
int main()
{
    using namespace std;

    vector<Review> books;
    Review temp;
    while (FillReview(temp))
        books.push_back(temp);
    if (books.size() > 0)
    {
        cout << "Thank you. You entered the following "
             << books.size() << " ratings:\n"
              << "Rating\tBook\n";
        for_each(books.begin(), books.end(), ShowReview);

        sort(books.begin(), books.end());
        cout << "Sorted by title:\nRating\tBook\n";
        for_each(books.begin(), books.end(), ShowReview);

        sort(books.begin(), books.end(), worseThan);
        cout << "Sorted by rating:\nRating\tBook\n";
        for_each(books.begin(), books.end(), ShowReview);

        random_shuffle(books.begin(), books.end());
        cout << "After shuffling:\nRating\tBook\n";
        for_each(books.begin(), books.end(), ShowReview);
    }
    else
        cout << "No entries. ";
```

```
    cout << "Bye.\n";
    return 0;
}

bool operator<(const Review & r1, const Review & r2)
{
    if (r1.title < r2.title)
        return true;
    else if (r1.title == r2.title && r1.rating < r2.rating)
        return true;
    else
        return false;
}

bool worseThan(const Review & r1, const Review & r2)
{
    if (r1.rating < r2.rating)
        return true;
    else
        return false;
}

bool FillReview(Review & rr)
{
    std::cout << "Enter book title (quit to quit): ";
    std::getline(std::cin,rr.title);
    if (rr.title == "quit")
        return false;
    std::cout << "Enter book rating: ";
    std::cin >> rr.rating;
    if (!std::cin)
        return false;
    // get rid of rest of input line
    while (std::cin.get() != '\n')
        continue;
    return true;
}

void ShowReview(const Review & rr)
{
    std::cout << rr.rating << "\t" << rr.title << std::endl;
}
```

程序清单 16.9 中程序的运行情况如下：

```
Enter book title (quit to quit): The Cat Who Can Teach You Weight Loss
Enter book rating: 8
Enter book title (quit to quit): The Dogs of Dharma
Enter book rating: 6
Enter book title (quit to quit): The Wimps of Wonk
Enter book rating: 3
Enter book title (quit to quit): Farewell and Delete
Enter book rating: 7
Enter book title (quit to quit): quit
Thank you. You entered the following 4 ratings:
Rating  Book
```

```
8       The Cat Who Can Teach You Weight Loss
6       The Dogs of Dharma
3       The Wimps of Wonk
7       Farewell and Delete
Sorted by title:
Rating  Book
7       Farewell and Delete
8       The Cat Who Can Teach You Weight Loss
6       The Dogs of Dharma
3       The Wimps of Wonk
Sorted by rating:
Rating  Book
3       The Wimps of Wonk
6       The Dogs of Dharma
7       Farewell and Delete
8       The Cat Who Can Teach You Weight Loss
After shuffling:
Rating  Book
7       Farewell and Delete
3       The Wimps of Wonk
6       The Dogs of Dharma
8       The Cat Who Can Teach You Weight Loss
Bye.
```

16.3.4 基于范围的 for 循环（C++11）

第 5 章说过，基于范围的 for 循环是为用于 STL 而设计的。为复习该循环，下面是第 5 章的第一个示例：

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

在这种 for 循环中，括号内的代码声明一个类型与容器存储的内容相同的变量，然后指出了容器的名称。接下来，循环体使用指定的变量依次访问容器的每个元素。例如，对于下述摘自程序清单 16.9 的语句：

```
for_each(books.begin(), books.end(), ShowReview);
```

可将其替换为下述基于范围的 for 循环：

```
for (auto x : books) ShowReview(x);
```

根据 book 的类型（vector<Review>），编译器将推断出 x 的类型为 Review，而循环将依次将 books 中的每个 Review 对象传递给 ShowReview( )。

不同于 for_each( )，基于范围的 for 循环可修改容器的内容，诀窍是指定一个引用参数。例如，假设有如下函数：

```
void InflateReview(Review &r){r.rating++;}
```

可使用如下循环对 books 的每个元素执行该函数：

```
for (auto & x : books) InflateReview(x);
```

16.4 泛型编程

有了一些使用 STL 的经验后，来看一看底层理念。STL 是一种泛型编程（generic programming）。面向

对象编程关注的是编程的数据方面，而泛型编程关注的是算法。它们之间的共同点是抽象和创建可重用代码，但它们的理念绝然不同。

泛型编程旨在编写独立于数据类型的代码。在 C++中，完成通用程序的工具是模板。当然，模板使得能够按泛型定义函数或类，而 STL 通过通用算法更进了一步。模板让这一切成为可能，但必须对元素进行仔细地设计。为解模板和设计是如何协同工作的，来看一看需要迭代器的原因。

16.4.1　为何使用迭代器

理解迭代器是理解 STL 的关键所在。模板使得算法独立于存储的数据类型，而迭代器使算法独立于使用的容器类型。因此，它们都是 STL 通用方法的重要组成部分。

为了解为何需要迭代器，我们来看如何为两种不同数据表示实现 find 函数，然后来看如何推广这种方法。首先看一个在 double 数组中搜索特定值的函数，可以这样编写该函数：

```
double * find_ar(double * ar, int n, const double & val)
{
    for (int i = 0; i < n; i++)
        if (ar[i] == val)
            return &ar[i];
    return 0;  // or, in C++11, return nullptr;
}
```

如果函数在数组中找到这样的值，则返回该值在数组中的地址，否则返回一个空指针。该函数使用下标来遍历数组。可以用模板将这种算法推广到包含= =运算符的、任意类型的数组。尽管如此，这种算法仍然与一种特定的数据结构（数组）关联在一起。

下面来看搜索另一种数据结构——链表的情况（第 12 章使用链表实现了 Queue 类）。链表由链接在一起的 Node 结构组成：

```
struct Node
{
    double item;
    Node * p_next;
};
```

假设有一个指向链表第一个节点的指针，每个节点的 p_next 指针都指向下一个节点，链表最后一个节点的 p_next 指针被设置为 0，则可以这样编写 find_ll( )函数：

```
Node* find_ll(Node * head, const double & val)
{
    Node * start;
    for (start = head; start!= 0; start = start->p_next)
        if (start->item == val)
            return start;
    return 0;
}
```

同样，也可以使用模板将这种算法推广到支持= =运算符的任何数据类型的链表。然而，这种算法也是与特定的数据结构——链表关联在一起。

从实现细节上看，这两个 find 函数的算法是不同的：一个使用数组索引来遍历元素，另一个则将 start 重置为 start->p_next。但从广义上说，这两种算法是相同的：将值依次与容器中的每个值进行比较，直到找到匹配的为止。

泛型编程旨在使用同一个 find 函数来处理数组、链表或任何其他容器类型。即函数不仅独立于容器中存储的数据类型，而且独立于容器本身的数据结构。模板提供了存储在容器中的数据类型的通用表示，因此还需要遍历容器中的值的通用表示，迭代器正是这样的通用表示。

要实现 find 函数，迭代器应具备哪些特征呢？下面是一个简短的列表。

- 应能够对迭代器执行解除引用的操作，以便能够访问它引用的值。即如果 p 是一个迭代器，则应对*p 进行定义。
- 应能够将一个迭代器赋给另一个。即如果 p 和 q 都是迭代器，则应对表达式 p=q 进行定义。
- 应能够将一个迭代器与另一个进行比较，看它们是否相等。即如果 p 和 q 都是迭代器，则应对 p= =q 和 p!=q 进行定义。
- 应能够使用迭代器遍历容器中的所有元素，这可以通过为迭代器 p 定义++p 和 p++来实现。

迭代器也可以完成其他的操作，但有上述功能就足够了，至少对于 find 函数是如此。实际上，STL 按功能的强弱定义了多种级别的迭代器，这将在后面介绍。顺便说一句，常规指针就能满足迭代器的要求，因此，可以这样重新编写 find_arr( )函数：

```
typedef double * iterator;
iterator find_ar(iterator ar, int n, const double & val)
{
    for (int i = 0; i < n; i++, ar++)
        if (*ar == val)
            return ar;
    return 0;
}
```

然后可以修改函数参数，使之接受两个指示区间的指针参数，其中的一个指向数组的起始位置，另一个指向数组的超尾（程序清单 7.8 与此类似）；同时函数可以通过返回尾指针，来指出没有找到要找的值。下面的 find_ar( )版本完成了这些修改：

```
typedef double * iterator;
iterator find_ar(iterator begin, iterator end, const double & val)
{
    iterator ar;
    for (ar = begin; ar != end;  ar++)
        if (*ar == val)
            return ar;
    return end;   // indicates val not found
}
```

对于 find_ll( )函数，可以定义一个迭代器类，其中定义了运算符*和++：

```
struct Node
{
    double item;
    Node * p_next;
};

class iterator
{
    Node * pt;
public:
    iterator() : pt(0) {}
    iterator (Node * pn) : pt(pn) {}
    double operator*() { return pt->item;}
    iterator& operator++()     // for ++it
    {
        pt = pt->p_next;
        return *this;
    }
    iterator operator++(int)  // for it++
    {
```

```
        iterator tmp = *this;
        pt = pt->p_next;
        return tmp;
    }
// ... operator==(), operator!=(), etc.
};
```

为区分++运算符的前缀版本和后缀版本，C++将 operator++作为前缀版本，将 operator++（int）作为后缀版本；其中的参数永远也不会被用到，所以不必指定其名称。

这里重点不是如何定义 iterator 类，而是有了这样的类后，第二个 find 函数就可以这样编写：

```
iterator find_ll(iterator head, const double & val)
{
    iterator start;
    for (start = head; start!= 0; ++start)
        if (*start == val)
            return start;
    return 0;
}
```

这和 find_ar( )几乎相同，差别在于如何谓词已到达最后一个值。find_ar( )函数使用超尾迭代器，而 find_ll( )使用存储在最后一个节点中的空值。除了这种差别外，这两个函数完全相同。例如，可以要求链表的最后一个元素后面还有一个额外的元素，即让数组和链表都有超尾元素，并在迭代器到达超尾位置时结束搜索。这样，find_ar( )和 find_ll( )检测数据尾的方式将相同，从而成为相同的算法。注意，增加超尾元素后，对迭代器的要求变成了对容器类的要求。

STL 遵循上面介绍的方法。首先，每个容器类（vector、list、deque 等）定义了相应的迭代器类型。对于其中的某个类，迭代器可能是指针；而对于另一个类，则可能是对象。不管实现方式如何，迭代器都将提供所需的操作，如*和++（有些类需要的操作可能比其他类多）。其次，每个容器类都有一个超尾标记，当迭代器递增到超越容器的最后一个值后，这个值将被赋给迭代器。每个容器类都有 begin( )和 end( )方法，它们分别返回一个指向容器的第一个元素和超尾位置的迭代器。每个容器类都使用++操作，让迭代器从指向第一个元素逐步指向超尾位置，从而遍历容器中的每一个元素。

使用容器类时，无需知道其迭代器是如何实现的，也无需知道超尾是如何实现的，而只需知道它有迭代器，其 begin( )返回一个指向第一个元素的迭代器，end( )返回一个指向超尾位置的迭代器即可。例如，假设要打印 vector<double>对象中的值，则可以这样做：

```
vector<double>::iterator pr;
for (pr = scores.begin(); pr != scores.end(); pr++)
    cout << *pr << endl;
```

其中，下面的代码行将 pr 的类型声明为 vector<double>类的迭代器：

```
vector<double> class:
vector<double>::iterator pr;
```

如果要使用 list<double>类模板来存储分数，则代码如下：

```
list<double>::iterator pr;
for (pr = scores.begin(); pr != scores.end(); pr++)
    cout << *pr << endl;
```

唯一不同的是 pr 的类型。因此，STL 通过为每个类定义适当的迭代器，并以统一的风格设计类，能够对内部表示绝然不同的容器，编写相同的代码。

使用 C++11 新增的自动类型推断可进一步简化：对于矢量或列表，都可使用如下代码：

```
for (auto pr = scores.begin(); pr != scores.end(); pr++)
    cout << *pr << endl;
```

实际上，作为一种编程风格，最好避免直接使用迭代器，而应尽可能使用 STL 函数（如 for_each( )）来处理细节。也可使用 C++11 新增的基于范围的 for 循环：

```
for (auto x : scores) cout << x << endl;
```

来总结一下 STL 方法。首先是处理容器的算法，应尽可能用通用的术语来表达算法，使之独立于数据类型和容器类型。为使通用算法能够适用于具体情况，应定义能够满足算法需求的迭代器，并把要求加到容器设计上。即基于算法的要求，设计基本迭代器的特征和容器特征。

16.4.2 迭代器类型

不同的算法对迭代器的要求也不同。例如，查找算法需要定义++运算符，以便迭代器能够遍历整个容器；它要求能够读取数据，但不要求能够写数据（它只是查看数据，而并不修改数据）。而排序算法要求能够随机访问，以便能够交换两个不相邻的元素。如果 iter 是一个迭代器，则可以通过定义+运算符来实现随机访问，这样就可以使用像 iter + 10 这样的表达式了。另外，排序算法要求能够读写数据。

STL 定义了 5 种迭代器，并根据所需的迭代器类型对算法进行了描述。这 5 种迭代器分别是输入迭代器、输出迭代器、正向迭代器、双向迭代器和随机访问迭代器。例如，find( )的原型与下面类似：

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value);
```

这指出，这种算法需要一个输入迭代器。同样，下面的原型指出排序算法需要一个随机访问迭代器：

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

对于这 5 种迭代器，都可以执行解除引用操作（即为它们定义了*运算符），也可进行比较，看其是相等（使用= =运算符，可能被重载了）还是不相等（使用!=运算符，可能被重载了）。如果两个迭代器相同，则对它们执行解除引用操作得到的值将相同。即如果表达式 iter1 == iter2 为真，则下述表达式也为真：

```
iter1 == iter2
```

is true, then the following is also true:

```
*iter1 == *iter2
```

当然，对于内置运算符和指针来说，情况也是如此。因此，这些要求将指导您如何对迭代器类重载这些运算符。下面来看迭代器的其他特征。

1. 输入迭代器

术语“输入”是从程序的角度说的，即来自容器的信息被视为输入，就像来自键盘的信息对程序来说是输入一样。因此，输入迭代器可被程序用来读取容器中的信息。具体地说，对输入迭代器解除引用将使程序能够读取容器中的值，但不一定能让程序修改值。因此，需要输入迭代器的算法将不会修改容器中的值。

输入迭代器必须能够访问容器中所有的值，这是通过支持++运算符（前缀格式和后缀格式）来实现的。如果将输入迭代器设置为指向容器中的第一个元素，并不断将其递增，直到到达超尾位置，则它将依次指向容器中的每一个元素。顺便说一句，并不能保证输入迭代器第二次遍历容器时，顺序不变。另外，输入迭代器被递增后，也不能保证其先前的值仍然可以被解除引用。基于输入迭代器的任何算法都应当是单通行（single-pass）的，不依赖于前一次遍历时的迭代器值，也不依赖于本次遍历中前面的迭代器值。

注意，输入迭代器是单向迭代器，可以递增，但不能倒退。

2. 输出迭代器

STL 使用术语“输出”来指用于将信息从程序传输给容器的迭代器，因此程序的输出就是容器的输入。输出迭代器与输入迭代器相似，只是解除引用让程序能修改容器值，而不能读取。也许您会感到奇怪，能够写，却不能读。发送到显示器上的输出就是如此，cout 可以修改发送到显示器的字符流，却不能读取屏

幕上的内容。STL 足够通用，其容器可以表示输出设备，因此容器也可能如此。另外，如果算法不用读取作容器的内容就可以修改它（如通过生成要存储的新值），则没有理由要求它使用能够读取内容的迭代器。

简而言之，对于单通行、只读算法，可以使用输入迭代器；而对于单通行、只写算法，则可以使用输出迭代器。

3. 正向迭代器

与输入迭代器和输出迭代器相似，正向迭代器只使用++运算符来遍历容器，所以它每次沿容器向前移动一个元素；然而，与输入和输出迭代器不同的是，它总是按相同的顺序遍历一系列值。另外，将正向迭代器递增后，仍然可以对前面的迭代器值解除引用（如果保存了它），并可以得到相同的值。这些特征使得多次通行算法成为可能。

正向迭代器既可以使得能够读取和修改数据，也可以使得只能读取数据：

```
int * pirw;        // read-write iterator
const int * pir;   // read-only iterator
```

4. 双向迭代器

假设算法需要能够双向遍历容器，情况将如何呢？例如，reverse 函数可以交换第一个元素和最后一个元素、将指向第一个元素的指针加 1、将指向第二个元素的指针减 1，并重复这种处理过程。双向迭代器具有正向迭代器的所有特性，同时支持两种（前缀和后缀）递减运算符。

5. 随机访问迭代器

有些算法（如标准排序和二分检索）要求能够直接跳到容器中的任何一个元素，这叫做随机访问，需要随机访问迭代器。随机访问迭代器具有双向迭代器的所有特性，同时添加了支持随机访问的操作（如指针增加运算）和用于对元素进行排序的关系运算符。表 16.3 列出了除双向迭代器的操作外，随机访问迭代器还支持的操作。其中，X 表示随机迭代器类型，T 表示被指向的类型，a 和 b 都是迭代器值，n 为整数，r 为随机迭代器变量或引用。

表 16.3　随机访问迭代器操作

表　达　式	描　　述
a + n	指向 a 所指向的元素后的第 n 个元素
n + a	与 a + n 相同
a - n	指向 a 所指向的元素前的第 n 个元素
r += n	等价于 r = r + n
r -= n	等价于 r = r – n
a[n]	等价于*(a + n)
b - a	结果为这样的 n 值，即 b = a + n
a < b	如果 b – a > 0，则为真
a > b	如果 b < a，则为真
a >= b	如果 !(a < b)，则为真
a <= b	如果 !(a > b)，则为真

像 a+n 这样的表达式仅当 a 和 a+n 都位于容器区间（包括超尾）内时才合法，

16.4.3　迭代器层次结构

您可能已经注意到，迭代器类型形成了一个层次结构。正向迭代器具有输入迭代器和输出迭代器的全部功能，同时还有自己的功能；双向迭代器具有正向迭代器的全部功能，同时还有自己的功能；随机访问迭代器具有正向迭代器的全部功能，同时还有自己的功能。表 16.4 总结了主要的迭代器功能。其中，i 为

迭代器，n 为整数。

表 16.4 迭代器性能

迭代器功能	输　入	输　出	正　向	双　向	随机访问
解除引用读取	有	无	有	有	有
解除引用写入	无	有	有	有	有
固定和可重复排序	无	无	有	有	有
++i　i++	有	有	有	有	有
--i　i--	无	无	无	有	有
i[n]	无	无	无	无	有
i + n	无	无	无	无	有
i - n	无	无	无	无	有
i += n	无	无	无	无	有
i -= n	无	无	无	无	有

根据特定迭代器类型编写的算法可以使用该种迭代器，也可以使用具有所需功能的任何其他迭代器。所以具有随机访问迭代器的容器可以使用为输入迭代器编写的算法。

为何需要这么多迭代器呢？目的是为了在编写算法尽可能使用要求最低的迭代器，并让它适用于容器的最大区间。这样，通过使用级别最低的输入迭代器，find( )函数便可用于任何包含可读取值的容器。而 sort( )函数由于需要随机访问迭代器，所以只能用于支持这种迭代器的容器。

注意，各种迭代器的类型并不是确定的，而只是一种概念性描述。正如前面指出的，每个容器类都定义了一个类级 typedef 名称——iterator，因此 vector<int>类的迭代器类型为 vector<int> :: interator。然而，该类的文档将指出，矢量迭代器是随机访问迭代器，它允许使用基于任何迭代器类型的算法，因为随机访问迭代器具有所有迭代器的功能。同样，list<int>类的迭代器类型为 list<int> :: iterator。STL 实现了一个双向链表，它使用双向迭代器，因此不能使用基于随机访问迭代器的算法，但可以使用基于要求较低的迭代器的算法。

16.4.4 概念、改进和模型

STL 有若干个用 C++语言无法表达的特性，如迭代器种类。因此，虽然可以设计具有正向迭代器特征的类，但不能让编译器将算法限制为只使用这个类。原因在于，正向迭代器是一系列要求，而不是类型。所设计的迭代器类可以满足这种要求，常规指针也能满足这种要求。STL 算法可以使用任何满足其要求的迭代器实现。STL 文献使用术语概念（concept）来描述一系列的要求。因此，存在输入迭代器概念、正向迭代器概念，等等。顺便说一句，如果所设计的容器类需要迭代器，可考虑 STL，它包含用于标准种类的迭代器模板。

概念可以具有类似继承的关系。例如，双向迭代器继承了正向迭代器的功能。然而，不能将 C++继承机制用于迭代器。例如，可以将正向迭代器实现为一个类，而将双向迭代器实现为一个常规指针。因此，对 C++而言，这种双向迭代器是一种内置类型，不能从类派生而来。然而，从概念上看，它确实能够继承。有些 STL 文献使用术语改进（refinement）来表示这种概念上的继承，因此，双向迭代器是对正向迭代器概念的一种改进。

概念的具体实现被称为模型（model）。因此，指向 int 的常规指针是一个随机访问迭代器模型，也是一个正向迭代器模型，因为它满足该概念的所有要求。

1. 将指针用作迭代器

迭代器是广义指针，而指针满足所有的迭代器要求。迭代器是 STL 算法的接口，而指针是迭代器，因此 STL 算法可以使用指针来对基于指针的非 STL 容器进行操作。例如，可将 STL 算法用于数组。假设

Receipts 是一个 double 数组，并要按升序对它进行排序：

```
const int SIZE = 100;
double Receipts[SIZE];
```

STL sort( )函数接受指向容器第一个元素的迭代器和指向超尾的迭代器作为参数。&Receipts[0]（或 Receipts）是第一个元素的地址，&Receipts[SIZE]（或 Receipts + SIZE）是数组最后一个元素后面的元素的地址。因此，下面的函数调用对数组进行排序：

```
sort(Receipts, Receipts + SIZE);
```

C++确保了表达式 Receipts + n 是被定义的，只要该表达式的结果位于数组中。因此，C++支持将超尾概念用于数组，使得可以将 STL 算法用于常规数组。由于指针是迭代器，而算法是基于迭代器的，这使得可将 STL 算法用于常规数组。同样，可以将 STL 算法用于自己设计的数组形式，只要提供适当的迭代器（可以是指针，也可以是对象）和超尾指示器即可。

copy( )、ostream_iterator 和 istream_iterator

STL 提供了一些预定义迭代器。为了解其中的原因，这里先介绍一些背景知识。有一种算法（名为 copy( )）可以将数据从一个容器复制到另一个容器中。这种算法是以迭代器方式实现的，所以它可以从一种容器到另一种容器进行复制，甚至可以在数组之间复制，因为可以将指向数组的指针用作迭代器。例如，下面的代码将一个数组复制到一个矢量中：

```
int casts[10] = {6, 7, 2, 9 ,4 , 11, 8, 7, 10, 5};
vector<int> dice[10];
copy(casts, casts + 10, dice.begin());   // copy array to vector
```

copy( )的前两个迭代器参数表示要复制的范围，最后一个迭代器参数表示要将第一个元素复制到什么位置。前两个参数必须是（或最好是）输入迭代器，最后一个参数必须是（或最好是）输出迭代器。Copy( )函数将覆盖目标容器中已有的数据，同时目标容器必须足够大，以便能够容纳被复制的元素。因此，不能使用 copy( )将数据放到空矢量中——至少，如果不采用本章后面将介绍的技巧，则不能这样做。

现在，假设要将信息复制到显示器上。如果有一个表示输出流的迭代器，则可以使用 copy( )。STL 为这种迭代器提供了 ostream_iterator 模板。用 STL 的话说，该模板是输出迭代器概念的一个模型，它也是一个适配器（adapter）——一个类或函数，可以将一些其他接口转换为 STL 使用的接口。可以通过包含头文件 iterator（以前为 iterator.h）并作下面的声明来创建这种迭代器：

```
#include <iterator>
...
ostream_iterator<int, char> out_iter(cout, " ");
```

out_iter 迭代器现在是一个接口，让您能够使用 cout 来显示信息。第一个模板参数（这里为 int）指出了被发送给输出流的数据类型；第二个模板参数（这里为 char）指出了输出流使用的字符类型（另一个可能的值是 wchar_t）。构造函数的第一个参数（这里为 cout）指出了要使用的输出流，它也可以是用于文件输出的流（参见第 17 章）；最后一个字符串参数是在发送给输出流的每个数据项后显示的分隔符。

可以这样使用迭代器：

```
*out_iter++ = 15;   // works like cout << 15 << " ";
```

对于常规指针，这意味着将 15 赋给指针指向的位置，然后将指针加 1。但对于该 ostream_iterator，这意味着将 15 和由空格组成的字符串发送到 cout 管理的输出流中，并为下一个输出操作做好了准备。可以将 copy( )用于迭代器，如下所示：

```
copy(dice.begin(), dice.end(), out_iter);    // copy vector to output stream
```

这意味着将 dice 容器的整个区间复制到输出流中，即显示容器的内容。

也可以不创建命名的迭代器，而直接构建一个匿名迭代器。即可以这样使用适配器：

```
copy(dice.begin(), dice.end(), ostream_iterator<int, char>(cout, " ") );
```

iterator 头文件还定义了一个 istream_iterator 模板，使 istream 输入可用作迭代器接口。它是一个输入迭

代器概念的模型，可以使用两个 istream_iterator 对象来定义 copy( )的输入范围：

```
copy(istream_iterator<int, char>(cin),
     istream_iterator<int, char>(), dice.begin());
```

与 ostream_iterator 相似，istream_iterator 也使用两个模板参数。第一个参数指出要读取的数据类型，第二个参数指出输入流使用的字符类型。使用构造函数参数 cin 意味着使用由 cin 管理的输入流，省略构造函数参数表示输入失败，因此上述代码从输入流中读取，直到文件结尾、类型不匹配或出现其他输入故障为止。

2. 其他有用的迭代器

除了 ostream_iterator 和 istream_iterator 之外，头文件 iterator 还提供了其他一些专用的预定义迭代器类型。它们是 reverse_iterator、back_insert_iterator、front_insert_iterator 和 insert_iterator。

我们先来看 reverse_iterator 的功能。对 reverse_iterator 执行递增操作将导致它被递减。为什么不直接对常规迭代器进行递减呢？主要原因是为了简化对已有的函数的使用。假设要显示 dice 容器的内容，正如刚才介绍的，可以使用 copy( )和 ostream_iterator 来将内容复制到输出流中：

```
ostream_iterator<int, char> out_iter(cout, " ");
copy(dice.begin(), dice.end(), out_iter);  // display in forward order
```

现在假设要反向打印容器的内容（可能您正在从事时间反演研究）。有很多方法都不管用，但与其在这里耽误工夫，不如来看看能够完成这种任务的方法。vector 类有一个名为 rbegin( )的成员函数和一个名为 rend( )的成员函数，前者返回一个指向超尾的反向迭代器，后者返回一个指向第一个元素的反向迭代器。因为对迭代器执行递增操作将导致它被递减，所以可以使用下面的语句来反向显示内容：

```
copy(dice.rbegin(), dice.rend(), out_iter); // display in reverse order
```

甚至不必声明反向迭代器。

注意：rbegin( )和 end( )返回相同的值（超尾），但类型不同（reverse_iterator 和 iterator）。同样，rend( )和 begin( )也返回相同的值（指向第一个元素的迭代器），但类型不同。

必须对反向指针做一种特殊补偿。假设 rp 是一个被初始化为 dice.rbegin( )的反转指针。那么*rp 是什么呢？因为 rbegin( )返回超尾，因此不能对该地址进行解除引用。同样，如果 rend( )是第一个元素的位置，则 copy( )必须提早一个位置停止，因为区间的结尾处不包括在区间中。

反向指针通过先递减，再解除引用解决了这两个问题。即*rp 将在*rp 的当前值之前对迭代器执行解除引用。也就是说，如果 rp 指向位置 6，则*rp 将是位置 5 的值，依次类推。程序清单 16.10 演示了如何使用 copy( )、istream 迭代器和反向迭代器。

程序清单 16.10　copyit.cpp

```
// copyit.cpp -- copy() and iterators
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
    using namespace std;

    int casts[10] = {6, 7, 2, 9 ,4 , 11, 8, 7, 10, 5};
    vector<int> dice(10);
    // copy from array to vector
    copy(casts, casts + 10, dice.begin());
    cout << "Let the dice be cast!\n";
```

```
    // create an ostream iterator
    ostream_iterator<int, char> out_iter(cout, " ");
    // copy from vector to output
    copy(dice.begin(), dice.end(), out_iter);
    cout << endl;
    cout <<"Implicit use of reverse iterator.\n";
    copy(dice.rbegin(), dice.rend(), out_iter);
    cout << endl;
    cout <<"Explicit use of reverse iterator.\n";
    vector<int>::reverse_iterator ri;
    for (ri = dice.rbegin(); ri != dice.rend(); ++ri)
        cout << *ri << ' ';
    cout << endl;

    return 0;
}
```

程序清单 16.10 中程序的输出如下：

```
Let the dice be cast!
6 7 2 9 4 11 8 7 10 5
Implicit use of reverse iterator.
5 10 7 8 11 4 9 2 7 6
Explicit use of reverse iterator.
5 10 7 8 11 4 9 2 7 6
```

如果可以在显式声明迭代器和使用 STL 函数来处理内部问题（如通过将 rbegin( )返回值传递给函数）之间选择，请采用后者。后一种方法要做的工作较少，人为出错的机会也较少。

另外三种迭代器（back_insert_iterator、front_insert_iterator 和 insert_iterator）也将提高 STL 算法的通用性。很多 STL 函数都与 copy( )相似，将结果发送到输出迭代器指示的位置。前面说过，下面的语句将值复制到从 dice.begin( )开始的位置：

```
copy(casts, casts + 10, dice.begin());
```

这些值将覆盖 dice 中以前的内容，且该函数假设 dice 有足够的空间，能够容纳这些值，即 copy( )不能自动根据发送值调整目标容器的长度。程序清单 16.10 考虑到了这种情况，将 dice 声明为包含 10 个元素。然而，如果预先并不知道 dice 的长度，该如何办呢？或者要将元素添加到 dice 中，而不是覆盖已有的内容，又该如何办呢？

三种插入迭代器通过将复制转换为插入解决了这些问题。插入将添加新的元素，而不会覆盖已有的数据，并使用自动内存分配来确保能够容纳新的信息。back_insert_iterator 将元素插入到容器尾部，而 front_insert_iterator 将元素插入到容器的前端。最后，insert_iterator 将元素插入到 insert_iterator 构造函数的参数指定的位置前面。这三个插入迭代器都是输出容器概念的模型。

这里存在一些限制。back_insert_iterator 只能用于允许在尾部快速插入的容器（快速插入指的是一个时间固定的算法，将在本章后面的“容器概念”一节做进一步讨论），vector 类符合这种要求。front_insert_iterator 只能用于允许在起始位置做时间固定插入的容器类型，vector 类不能满足这种要求，但 queue 满足。insert_iterator 没有这些限制，因此可以用它把信息插入到矢量的前端。然而，front_insert_iterator 对于那些支持它的容器来说，完成任务的速度更快。

提示： *可以用 insert_iterator 将复制数据的算法转换为插入数据的算法。*

这些迭代器将容器类型作为模板参数，将实际的容器标识符作为构造函数参数。也就是说，要为名为 dice 的 vector<int>容器创建一个 back_insert_iterator，可以这样做：

```
back_insert_iterator<vector<int> > back_iter(dice);
```

必须声明容器类型的原因是，迭代器必须使用合适的容器方法。back_insert_iterator 的构造函数将假设

传递给它的类型有一个 push_back( )方法。copy( )是一个独立的函数，没有重新调整容器大小的权限。但前面的声明让 back_iter 能够使用方法 vector<int>::push_back( )，该方法有这样的权限。

声明 front_insert_iterator 的方式与此相同。对于 insert_iterator 声明，还需一个指示插入位置的构造函数参数：

```
insert_iterator<vector<int> > insert_iter(dice, dice.begin() );
```

程序清单 16.11 演示了这两种迭代器的用法，还使用 for_each( )而不是 ostream 迭代器进行输出。

程序清单 16.11　inserts.cpp

```
// inserts.cpp -- copy() and insert iterators
#include <iostream>
#include <string>
#include <iterator>
#include <vector>
#include <algorithm>

void output(const std::string & s) {std::cout << s << " ";}

int main()
{
    using namespace std;
    string s1[4] = {"fine", "fish", "fashion", "fate"};
    string s2[2] = {"busy", "bats"};
    string s3[2] = {"silly", "singers"};
    vector<string> words(4);
    copy(s1, s1 + 4, words.begin());
    for_each(words.begin(), words.end(), output);
    cout << endl;
// construct anonymous back_insert_iterator object
    copy(s2, s2 + 2, back_insert_iterator<vector<string> >(words));
    for_each(words.begin(), words.end(), output);
    cout << endl;

// construct anonymous insert_iterator object
    copy(s3, s3 + 2, insert_iterator<vector<string> >(words,
                                                      words.begin()));
    for_each(words.begin(), words.end(), output);
    cout << endl;
    return 0;
}
```

程序清单 16.11 中程序的输出如下：

```
fine fish fashion fate
fine fish fashion fate busy bats
silly singers fine fish fashion fate busy bats
```

第一个 copy( )从 s1 中复制 4 个字符串到 words 中。这之所以可行，在某种程度上说是由于 words 被声明为能够存储 4 个字符串，这等于被复制的字符串数目。然后，back_insert_iterator 将 s2 中的字符串插入到 words 数组的末尾，将 words 的长度增加到 6 个元素。最后，insert_iterator 将 s3 中的两个字符串插入到 words 的第一个元素的前面，将 words 的长度增加到 8 个元素。如果程序试图使用 words.end( )和 words.begin( )作为迭代器，将 s2 和 s3 复制到 words 中，words 将没有空间来存储新数据，程序可能会由于内存违规而异常终止。

如果您被这些迭代器搞晕，则请记住，只要使用就会熟悉它们。另外还请记住，这些预定义迭代器提高了 STL 算法的通用性。因此，copy( )不仅可以将信息从一个容器复制到另一个容器，还可以将信息从容

器复制到输出流，从输入流复制到容器中。还可以使用 copy( )将信息插入到另一个容器中。因此使用同一个函数可以完成很多工作。copy( )只是是使用输出迭代器的若干 STL 函数之一，因此这些预定义迭代器也增加了这些函数的功能。

16.4.5　容器种类

STL 具有容器概念和容器类型。概念是具有名称（如容器、序列容器、关联容器等）的通用类别；容器类型是可用于创建具体容器对象的模板。以前的 11 个容器类型分别是 deque、list、queue、priority_queue、stack、vector、map、multimap、set、multiset 和 bitset（本章不讨论 bitset，它是在比特级处理数据的容器）；C++11 新增了 forward_list、unordered_map、unordered_multimap、unordered_set 和 unordered_multiset，且不将 bitset 视为容器，而将其视为一种独立的类别。因为概念对类型进行了分类，下面先讨论它们。

1. 容器概念

没有与基本容器概念对应的类型，但概念描述了所有容器类都通用的元素。它是一个概念化的抽象基类——说它概念化，是因为容器类并不真正使用继承机制。换句话说，容器概念指定了所有 STL 容器类都必须满足的一系列要求。

容器是存储其他对象的对象。被存储的对象必须是同一种类型的，它们可以是 OOP 意义上的对象，也可以是内置类型值。存储在容器中的数据为容器所有，这意味着当容器过期时，存储在容器中的数据也将过期（然而，如果数据是指针的话，则它指向的数据并不一定过期）。

不能将任何类型的对象存储在容器中，具体地说，类型必须是可复制构造的和可赋值的。基本类型满足这些要求；只要类定义没有将复制构造函数和赋值运算符声明为私有或保护的，则也满足这种要求。C++11 改进了这些概念，添加了术语可复制插入（CopyInsertable）和可移动插入（MoveInsertable），但这里只进行简单的概述。

基本容器不能保证其元素都按特定的顺序存储，也不能保证元素的顺序不变，但对概念进行改进后，则可以增加这样的保证。所有的容器都提供某些特征和操作。表 16.5 对一些通用特征进行了总结。其中，X 表示容器类型，如 vector；T 表示存储在容器中的对象类型；a 和 b 表示类型为 X 的值；r 表示类型为 X&的值；u 表示类型为 X 的标识符（即如果 X 表示 vector<int>，则 u 是一个 vector<int>对象）。

表 16.5　一些基本的容器特征

表　达　式	返 回 类 型	说　　明	复　杂　度
X :: iterator	指向 T 的迭代器类型	满足正向迭代器要求的任何迭代器	编译时间
X :: value_type	T	T 的类型	编译时间
X u;		创建一个名为 u 的空容器	固定
X( );		创建一个匿名的空容器	固定
X u(a);		调用复制构造函数后 u == a	线性
X u = a;		作用同 X u(a);	线性
r = a;	X&	调用赋值运算符后 r == a	线性
(&a)->~X( )	void	对容器中每个元素应用析构函数	线性
a.begin( )	迭代器	返回指向容器第一个元素的迭代器	固定
a.end( )	迭代器	返回超尾值迭代器	固定
a.size( )	无符号整型	返回元素个数，等价于 a.end( )– a.begin( )	固定
a.swap(b)	void	交换 a 和 b 的内容	固定
a = = b	可转换为 bool	如果 a 和 b 的长度相同，且 a 中每个元素都等于（= = 为真）b 中相应的元素，则为真	线性
a != b	可转换为 bool	返回!(a= =b)	线性

表 16.5 中的“复杂度”一列描述了执行操作所需的时间。这个表列出了 3 种可能性，从快到慢依次为：

- 编译时间；
- 固定时间；
- 线性时间。

如果复杂度为编译时间，则操作将在编译时执行，执行时间为 0。固定复杂度意味着操作发生在运行阶段，但独立于对象中的元素数目。线性复杂度意味着时间与元素数目成正比。即如果 a 和 b 都是容器，则 a = = b 具有线性复杂度，因为= =操作必须用于容器中的每个元素。实际上，这是最糟糕的情况。如果两个容器的长度不同，则不需要作任何的单独比较。

固定时间和线性时间复杂度

假设有一个装满大包裹的狭长盒子，包裹一字排开，而盒子只有一端是打开的。假设任务是从打开的一端取出一个包裹，则这将是一项固定时间任务。不管在打开的一端后面有 10 个还是 1000 个包裹，都没有区别。

现在假设任务是取出盒子中没有打开的一端的那个包裹，则这将是线性时间任务。如果盒子里有 10 个包裹，则必须取出 10 个包裹才能拿到封口端的那个包裹；如果有 100 个包裹，则必须取出 100 个包裹。假设是一个不知疲倦的工人来做，每次只能取出 1 个包裹，则需要取 10 次或更多。

现在假设任务是取出任意一个包裹，则可能取出第一个包裹。然而，通常必须移动的包裹数目仍旧与容器中包裹的数目成正比，所以这种任务依然是线性时间复杂度。

如果盒子各边都可打开，而不是狭长的，则这种任务的复杂度将是固定时间的，因为可以直接取出想要的包裹，而不用移动其他的包裹。

时间复杂度概念描述了容器长度对执行时间的影响，而忽略了其他因素。如果超人从一端打开的盒子中取出包裹的速度比普通人快 100 倍，则他完成任务时，复杂度仍然是线性时间的。在这种情况下，他取出封闭盒子中包裹（一端打开，复杂度为线性时间）的速度将比普通人取出开放盒子中包裹（复杂度为固定时间）要快，条件是盒子里没有太多的包裹。

复杂度要求是 STL 特征，虽然实现细节可以隐藏，但性能规格应公开，以便程序员能够知道完成特定操作的计算成本。

2. C++11 新增的容器要求

表 16.6 列出了 C++11 新增的通用容器要求。在这个表中，rv 表示类型为 X 的非常量右值，如函数的返回值。另外，在表 16.5 中，要求 X::iterator 满足正向迭代器的要求，而以前只要求它不是输出迭代器。

表 16.6　C++11 新增的基本容器要求

表 达 式	返 回 类 型	说　明	复 杂 度
X u(rv);		调用移动构造函数后，u 的值与 rv 的原始值相同	线性
X u = rv;		作用同 X u(rv);	
a = rv;	X&	调用移动赋值运算符后，u 的值与 rv 的原始值相同	线性
a.cbegin( )	const_iterator	返回指向容器第一个元素的 const 迭代器	固定
a.cend( )	const_iterator	返回超尾值 const 迭代器	固定

复制构造和复制赋值以及移动构造和移动赋值之间的差别在于，复制操作保留源对象，而移动操作可修改源对象，还可能转让所有权，而不做任何复制。如果源对象是临时的，移动操作的效率将高于常规复制。第 18 章将更详细地介绍移动语义。

3. 序列

可以通过添加要求来改进基本的容器概念。序列（sequence）是一种重要的改进，因为 7 种 STL

容器类型（deque、C++11 新增的 forward_list、list、queue、priority_queue、stack 和 vector）都是序列（本书前面说过，队列让您能够在队尾添加元素，在队首删除元素。deque 表示的双端队列允许在两端添加和删除元素）。序列概念增加了迭代器至少是正向迭代器这样的要求，这保证了元素将按特定顺序排列，不会在两次迭代之间发生变化。array 也被归类到序列容器，虽然它并不满足序列的所有要求。

序列还要求其元素按严格的线性顺序排列，即存在第一个元素、最后一个元素，除第一个元素和最后一个元素外，每个元素前后都分别有一个元素。数组和链表都是序列，但分支结构（其中每个节点都指向两个子节点）不是。

因为序列中的元素具有确定的顺序，因此可以执行诸如将值插入到特定位置、删除特定区间等操作。表 16.7 列出了这些操作以及序列必须完成的其他操作。该表格使用的表示法与表 16.5 相同，此外，t 表示类型为 T（存储在容器中的值的类型）的值，n 表示整数，p、q、i 和 j 表示迭代器。

表 16.7　序列的要求

表 达 式	返 回 类 型	说　明
X a(n, t);		声明一个名为 a 的由 n 个 t 值组成的序列
X(n, t)		创建一个由 n 个 t 值组成的匿名序列
X a(i, j)		声明一个名为 a 的序列，并将其初始化为区间[i，j)的内容
X(i, j)		创建一个匿名序列，并将其初始化为区间[i，j)的内容
a. insert(p, t)	迭代器	将 t 插入到 p 的前面
a.insert(p, n, t)	void	将 n 个 t 插入到 p 的前面
a.insert(p, i, j)	void	将区间[i，j)中的元素插入到 p 的前面
a.erase(p)	迭代器	删除 p 指向的元素
a.erase(p, q)	迭代器	删除区间[p，q)中的元素
a.clear( )	void	等价于 erase(begin( ), end( ))

因为模板类 deque、list、queue、priority_queue、stack 和 vector 都是序列概念的模型，所以它们都支持表 16.7 所示的运算符。除此之外，这 6 个模型中的一些还可使用其他操作。在允许的情况下，它们的复杂度为固定时间。表 16.8 列出了其他操作。

表 16.8　序列的可选要求

表 达 式	返 回 类 型	含　义	容　器
a.front( )	T&	*a.begin( )	vector、list、deque
a.back( )	T&	*- -a.end( )	vector、list、deque
a.push_front(t)	void	a.insert(a.begin( ), t)	list、deque
a.push_back(t)	void	a.insert(a.end( ), t)	vector、list、deque
a.pop_front(t)	void	a.erase(a.begin( ))	list、deque
a.pop_back(t)	void	a.erase(- -a.end( ))	vector、list、deque
a[n]	T&	*(a.begin( )+ n)	vector、deque
a.at(t)	T&	*(a.begin( )+ n)	vector、deque

表 16.8 有些需要说明的地方。首先，a[n]和 a.at(n)都返回一个指向容器中第 n 个元素（从 0 开始编号）的引用。它们之间的差别在于，如果 n 落在容器的有效区间外，则 a.at(n)将执行边界检查，并引发 out_of_range 异常。其次，可能有人会问，为何为 list 和 deque 定义了 push_front( )，而没有为 vector 定义？假设要将一个新值插入到包含 100 个元素的矢量的最前面。要腾出空间，必须将第 99 个元素移到位置 100，然后把第 98 个元素移动到位置 99，依此类推。这种操作的复杂度为线性时间，因为移动 100 个元素所需的时间为移

动单个元素的 100 倍。但表 16.8 的操作被假设为仅当其复杂度为固定时间时才被实现。链表和双端队列的设计允许将元素添加到前端，而不用移动其他元素，所以它们可以以固定时间的复杂度来实现 push_front( )。图 16.4 说明了 push_front( )和 push_back( )。

```
char word[4] = "cow";
deque<char>dword(word, word+3);
```

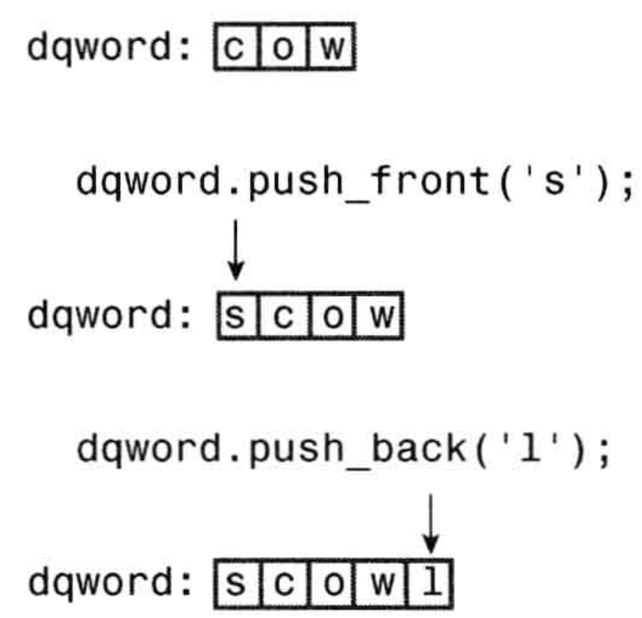

图 16.4 push_front( )和 push_back( )

下面详细介绍这 7 种序列容器类型。

（1）vector

前面介绍了多个使用 vector 模板的例子，该模板是在 vector 头文件中声明的。简单地说，vector 是数组的一种类表示，它提供了自动内存管理功能，可以动态地改变 vector 对象的长度，并随着元素的添加和删除而增大和缩小。它提供了对元素的随机访问。在尾部添加和删除元素的时间是固定的，但在头部或中间插入和删除元素的复杂度为线性时间。

除序列外，vector 还是可反转容器（reversible container）概念的模型。这增加了两个类方法：rbegin( )和 rend( )，前者返回一个指向反转序列的第一个元素的迭代器，后者返回反转序列的超尾迭代器。因此，如果 dice 是一个 vector<int>容器，而 Show(int)是显示一个整数的函数，则下面的代码将首先正向显示 dice 的内容，然后反向显示：

```
for_each(dice.begin(), dice.end(), Show);     // display in order
cout << endl;
for_each(dice.rbegin(), dice.rend(), Show);  // display in reversed order
cout << endl;
```

这两种方法返回的迭代器都是类级类型 reverse_iterator。对这样的迭代器进行递增，将导致它反向遍历可反转容器。

vector 模板类是最简单的序列类型，除非其他类型的特殊优点能够更好地满足程序的要求，否则应默认使用这种类型。

（2）deque

deque 模板类（在 deque 头文件中声明）表示双端队列（double-ended queue），通常被简称为 deque。在 STL 中，其实现类似于 vector 容器，支持随机访问。主要区别在于，从 deque 对象的开始位置插入和删除元素的时间是固定的，而不像 vector 中那样是线性时间的。所以，如果多数操作发生在序列的起始和结尾处，则应考虑使用 deque 数据结构。

为实现在 deque 两端执行插入和删除操作的时间为固定的这一目的，deque 对象的设计比 vector 对象更为复杂。因此，尽管二者都提供对元素的随机访问和在序列中部执行线性时间的插入和删除操作，但 vector 容器执行这些操作时速度要快些。

（3）list

list 模板类（在 list 头文件中声明）表示双向链表。除了第一个和最后一个元素外，每个元素都与前后的元素相链接，这意味着可以双向遍历链表。list 和 vector 之间关键的区别在于，list 在链表中任一位置进行插入和删除的时间都是固定的（vector 模板提供了除结尾处外的线性时间的插入和删除，在结尾处，它提供了固定时间的插入和删除）。因此，vector 强调的是通过随机访问进行快速访问，而 list 强调的是元素的快速插入和删除。

与 vector 相似，list 也是可反转容器。与 vector 不同的是，list 不支持数组表示法和随机访问。与矢量迭代器不同，从容器中插入或删除元素之后，链表迭代器指向元素将不变。我们来解释一下这句话。例如，假设有一个指向 vector 容器第 5 个元素的迭代器，并在容器的起始处插入一个元素。此时，必须移动其他所有元素，以便腾出位置，因此插入后，第 5 个元素包含的值将是以前第 4 个元素的值。因此，迭代器指向的位置不变，但数据不同。然后，在链表中插入新元素并不会移动已有的元素，而只是修改链接信息。指向某个元素的迭代器仍然指向该元素，但它链接的元素可能与以前不同。

除序列和可反转容器的函数外，list 模板类还包含了链表专用的成员函数。表 16.9 列出了其中一些（有关 STL 方法和函数的完整列表，请参见附录 G）。通常不必担心 Alloc 模板参数，因为它有默认值。

表 16.9　list 成员函数

函　数	说　明
void merge(list<T, Alloc>& x)	将链表 x 与调用链表合并。两个链表必须已经排序。合并后的经过排序的链表保存在调用链表中，x 为空。这个函数的复杂度为线性时间
void remove(const T & val)	从链表中删除 val 的所有实例。这个函数的复杂度为线性时间
void sort( )	使用<运算符对链表进行排序；N 个元素的复杂度为 NlogN
void splice(iterator pos, list<T, Alloc>x)	将链表 x 的内容插入到 pos 的前面，x 将为空。这个函数的的复杂度为固定时间
void unique( )	将连续的相同元素压缩为单个元素。这个函数的复杂度为线性时间

程序清单 16.12 演示了这些方法和 insert( )方法（所有模拟序列的 STL 类都有这种方法）的用法。

程序清单 16.12　list.cpp

```
// list.cpp -- using a list
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>

void outint(int n) {std::cout << n << " ";}

int main()
{
    using namespace std;
    list<int> one(5, 2); // list of 5 2s
    int stuff[5] = {1,2,4,8, 6};
    list<int> two;
    two.insert(two.begin(),stuff, stuff + 5 );
    int more[6] = {6, 4, 2, 4, 6, 5};
    list<int> three(two);
    three.insert(three.end(), more, more + 6);

    cout << "List one: ";
    for_each(one.begin(),one.end(), outint);
    cout << endl << "List two: ";
    for_each(two.begin(), two.end(), outint);
    cout << endl << "List three: ";
    for_each(three.begin(), three.end(), outint);
    three.remove(2);
    cout << endl << "List three minus 2s: ";
    for_each(three.begin(), three.end(), outint);
    three.splice(three.begin(), one);
    cout << endl << "List three after splice: ";
    for_each(three.begin(), three.end(), outint);
    cout << endl << "List one: ";
    for_each(one.begin(), one.end(), outint);
    three.unique();
    cout << endl << "List three after unique: ";
    for_each(three.begin(), three.end(), outint);
    three.sort();
```

```
    three.unique();
    cout << endl << "List three after sort & unique: ";
    for_each(three.begin(), three.end(), outint);
    two.sort();
    three.merge(two);
    cout << endl << "Sorted two merged into three: ";
    for_each(three.begin(), three.end(), outint);
    cout << endl;

    return 0;
}
```

下面是程序清单 16.12 中程序的输出：

```
List one: 2 2 2 2 2
List two: 1 2 4 8 6
List three: 1 2 4 8 6 6 4 2 4 6 5
List three minus 2s: 1 4 8 6 6 4 4 6 5
List three after splice: 2 2 2 2 2 1 4 8 6 6 4 4 6 5
List one:
List three after unique: 2 1 4 8 6 4 6 5
List three after sort & unique: 1 2 4 5 6 8
Sorted two merged into three: 1 1 2 2 4 4 5 6 6 8 8
```

（4）程序说明

程序清单 16.12 中程序使用了 for_each()算法和 outint( )函数来显示列表。在 C++11 中，也可使用基于范围的 for 循环：

```
for (auto x : three) cout << x << " ";
```

insert( )和 splice( )之间的主要区别在于：insert( )将原始区间的副本插入到目标地址，而 splice( )则将原始区间移到目标地址。因此，在 one 的内容与 three 合并后，one 为空。（splice( )方法还有其他原型，用于移动单个元素和元素区间）。splice( )方法执行后，迭代器仍有效。也就是说，如果将迭代器设置为指向 one 中的元素，则在 splice( )将它重新定位到元素 three 后，该迭代器仍然指向相同的元素。

注意，unique( )只能将相邻的相同值压缩为单个值。程序执行 three.unique( )后，three 中仍包含不相邻的两个 4 和两个 6。但应用 sort( )后再应用 unique( )时，每个值将只占一个位置。

还有非成员 sort( )函数（程序清单 16.9），但它需要随机访问迭代器。因为快速插入的代价是放弃随机访问功能，所以不能将非成员函数 sort( )用于链表。因此，这个类中包括了一个只能在类中使用的成员版本。

（5）list 工具箱

list 方法组成了一个方便的工具箱。例如，假设有两个邮件列表要整理，则可以对每个列表进行排序，合并它们，然后使用 unique( )来删除重复的元素。

sort( )、merge( )和 unique( )方法还各自拥有接受另一个参数的版本，该参数用于指定用来比较元素的函数。同样，remove( )方法也有一个接受另一个参数的版本，该参数用于指定用来确定是否删除元素的函数。这些参数都是谓词函数，将稍后介绍。

（6）forward_list（C++11）

C++11 新增了容器类 forward_list，它实现了单链表。在这种链表中，每个节点都只链接到下一个节点，而没有链接到前一个节点。因此 forward_list 只需要正向迭代器，而不需要双向迭代器。因此，不同于 vector 和 list，forward_list 是不可反转的容器。相比于 list，forward_list 更简单、更紧凑，但功能也更少。

（7）queue

queue 模板类（在头文件 queue（以前为 queue.h）中声明）是一个适配器类。由前所述，ostream_iterator 模板就是一个适配器，让输出流能够使用迭代器接口。同样，queue 模板让底层类（默认为 deque）展示典

型的队列接口。

queue 模板的限制比 deque 更多。它不仅不允许随机访问队列元素，甚至不允许遍历队列。它把使用限制在定义队列的基本操作上，可以将元素添加到队尾、从队首删除元素、查看队首和队尾的值、检查元素数目和测试队列是否为空。表 16.10 列出了这些操作。

表 16.10　queue 的操作

方　法	说　明
bool empty( )const	如果队列为空，则返回 true；否则返回 false
size_type size( )const	返回队列中元素的数目
T& front( )	返回指向队首元素的引用
T& back( )	返回指向队尾元素的引用
void push(const T& x)	在队尾插入 x
void pop( )	删除队首元素

注意，pop( )是一个删除数据的方法，而不是检索数据的方法。如果要使用队列中的值，应首先使用 front( )来检索这个值，然后使用 pop( )将它从队列中删除。

（8）priority_queue

priority_queue 模板类（在 queue 头文件中声明）是另一个适配器类，它支持的操作与 queue 相同。两者之间的主要区别在于，在 priority_queue 中，最大的元素被移到队首（生活不总是公平的，队列也一样）。内部区别在于，默认的底层类是 vector。可以修改用于确定哪个元素放到队首的比较方式，方法是提供一个可选的构造函数参数：

```
priority_queue<int> pq1;                // default version
priority_queue<int> pq2(greater<int>);  // use greater<int> to order
```

greater< >( )函数是一个预定义的函数对象，本章稍后将讨论它。

（9）stack

与 queue 相似，stack（在头文件 stack——以前为 stack.h——中声明）也是一个适配器类，它给底层类（默认情况下为 vector）提供了典型的栈接口。

stack 模板的限制比 vector 更多。它不仅不允许随机访问栈元素，甚至不允许遍历栈。它把使用限制在定义栈的基本操作上，即可以将压入推到栈顶、从栈顶弹出元素、查看栈顶的值、检查元素数目和测试栈是否为空。表 16.11 列出了这些操作。

表 16.11　stack 的操作

方　法	说　明
bool empty( )const	如果栈为空，则返回 true；否则返回 false
size_type size( )const	返回栈中的元素数目
T& top( )	返回指向栈顶元素的引用
void push(const T& x)	在栈顶部插入 x
void pop( )	删除栈顶元素

与 queue 相似，如果要使用栈中的值，必须首先使用 top( )来检索这个值，然后使用 pop( )将它从栈中删除。

（10）array（C++11）

第 4 章介绍过，模板类 array 是否头文件 array 中定义的，它并非 STL 容器，因为其长度是固定的。因此，array 没有定义调整容器大小的操作，如 push_back( )和 insert( )，但定义了对它来说有意义的成员函数，如 operator [] ()和 at( )。可将很多标准 STL 算法用于 array 对象，如 copy( )和 for_each( )。

16.4.4 关联容器

关联容器（associative container）是对容器概念的另一个改进。关联容器将值与键关联在一起，并使用键来查找值。例如，值可以是表示雇员信息（如姓名、地址、办公室号码、家庭电话和工作电话、健康计划等）的结构，而键可以是唯一的员工编号。为获取雇员信息，程序将使用键查找雇员结构。前面说过，对于容器 X，表达式 X::value_type 通常指出了存储在容器中的值类型。对于关联容器来说，表达式 X::key_type 指出了键的类型。

关联容器的优点在于，它提供了对元素的快速访问。与序列相似，关联容器也允许插入新元素，但不能指定元素的插入位置。原因是关联容器通常有用于确定数据放置位置的算法，以便能够快速检索信息。

关联容器通常是使用某种树实现的。树是一种数据结构，其根节点链接到一个或两个节点，而这些节点又链接到一个或两个节点，从而形成分支结构。像链表一样，节点使得添加或删除数据项比较简单；但相对于链表，树的查找速度更快。

STL 提供了 4 种关联容器：set、multiset、map 和 multimap。前两种是在头文件 set（以前分别为 set.h 和 multiset.h）中定义的，而后两种是在头文件 map（以前分别为 map.h 和 multimap.h）中定义的。

最简单的关联容器是 set，其值类型与键相同，键是唯一的，这意味着集合中不会有多个相同的键。确实，对于 set 来说，值就是键。multiset 类似于 set，只是可能有多个值的键相同。例如，如果键和值的类型为 int，则 multiset 对象包含的内容可以是 1、2、2、2、3、5、7、7。

在 map 中，值与键的类型不同，键是唯一的，每个键只对应一个值。multimap 与 map 相似，只是一个键可以与多个值相关联。

有关这些类型的信息很多，无法在本章全部列出（但附录 G 列出了方法），这里只介绍一个使用 set 的简单例子和一个使用 multimap 的简单例子。

1. set 示例

STL set 模拟了多个概念，它是关联集合，可反转，可排序，且键是唯一的，所以不能存储多个相同的值。与 vector 和 list 相似，set 也使用模板参数来指定要存储的值类型：

```
set<string> A;  // a set of string objects
```

第二个模板参数是可选的，可用于指示用来对键进行排序的比较函数或对象。默认情况下，将使用模板 less<>（稍后将讨论）。老式 C++实现可能没有提供默认值，因此必须显式指定模板参数：

```
set<string, less<string> > A;  // older implementation
```

请看下面的代码：

```
const int N = 6;
string s1[N] = {"buffoon", "thinkers", "for", "heavy", "can", "for"};
set<string> A(s1, s1 + N); // initialize set A using a range from array
ostream_iterator<string, char> out(cout, " ");
copy(A.begin(), A.end(), out);
```

与其他容器相似，set 也有一个将迭代器区间作为参数的构造函数（参见表 16.6）。这提供了一种将集合初始化为数组内容的简单方法。请记住，区间的最后一个元素是超尾，s1 + N 指向数组 s1 尾部后面的一个位置。上述代码片段的输出表明，键是唯一的（字符串“for”在数组中出现了 2 次，但在集合中只出现 1 次），且集合被排序：

```
buffoon can for heavy thinkers
```

数学为集合定义了一些标准操作，例如，并集包含两个集合合并后的内容。如果两个集合包含相同的值，则这个值将在并集中只出现一次，这是因为键是唯一的。交集包含两个集合都有的元素。两个集合的差是第一个集合减去两个集合都有的元素。

STL 提供了支持这些操作的算法。它们是通用函数，而不是方法，因此并非只能用于 set 对象。然而，所有 set 对象都自动满足使用这些算法的先决条件，即容器是经过排序的。set_union( )函数接受 5 个迭

代器参数。前两个迭代器定义了第一个集合的区间，接下来的两个定义了第二个集合区间，最后一个迭代器是输出迭代器，指出将结果集合复制到什么位置。例如，要显示集合 A 和 B 的并集，可以这样做：

```
set_union(A.begin(), A.end(), B.begin(), B.end(),
          ostream_iterator<string, char> out(cout, " "));
```

假设要将结果放到集合 C 中，而不是显示它，则最后一个参数应是一个指向 C 的迭代器。显而易见的选择是 C.begin( )，但它不管用，原因有两个。首先，关联集合将键看作常量，所以 C.begin( )返回的迭代器是常量迭代器，不能用作输出迭代器。不直接使用 C.begin( )的第二个原因是，与 copy( )相似，set_union( )将覆盖容器中已有的数据，并要求容器有足够的空间容纳新信息。C 是空的，不能满足这种要求。但前面讨论的模板 insert_iterator 可以解决这两个问题。前面说过，它可以将复制转换为插入。另外，它还模拟了输出迭代器概念，可以用它将信息写入容器。因此，可以创建一个匿名 insert_iterator，将信息复制给 C。前面说过，其构造函数将容器名称和迭代器作为参数：

```
set_union(A.begin(), A.end(), B.begin(), B.end(),
          insert_iterator<set<string> >(C, C.begin()));
```

函数 set_intersection( )和 set_difference( )分别查找交集和获得两个集合的差，它们的接口与 set_union( )相同。

两个有用的 set 方法是 lower_bound( )和 upper_bound( )。方法 lower_bound( )将键作为参数并返回一个迭代器，该迭代器指向集合中第一个不小于键参数的成员。同样，方法 upper_bound( )将键作为参数，并返回一个迭代器，该迭代器指向集合中第一个大于键参数的成员。例如，如果有一个字符串集合，则可以用这些方法获得一个这样的区间，即包含集合中从“b”到“f”的所有字符串。

因为排序决定了插入的位置，所以这种类包含只指定要插入的信息，而不指定位置的插入方法。例如，如果 A 和 B 是字符串集合，则可以这样做：

```
string s("tennis");
A.insert(s);                    // insert a value
B.insert(A.begin(), A.end());   // insert a range
```

程序清单 16.13 演示了集合的这些用途。

程序清单 16.13　setops.cpp

```
// setops.cpp -- some set operations
#include <iostream>
#include <string>
#include <set>
#include <algorithm>
#include <iterator>

int main()
{
    using namespace std;
    const int N = 6;
    string s1[N] = {"buffoon", "thinkers", "for", "heavy", "can", "for"};
    string s2[N] = {"metal", "any", "food", "elegant", "deliver","for"};

    set<string> A(s1, s1 + N);
    set<string> B(s2, s2 + N);

    ostream_iterator<string, char> out(cout, " ");
    cout << "Set A: ";
    copy(A.begin(), A.end(), out);
    cout << endl;
    cout << "Set B: ";
    copy(B.begin(), B.end(), out);
    cout << endl;
```

```
    cout << "Union of A and B:\n";
    set_union(A.begin(), A.end(), B.begin(), B.end(), out);
    cout << endl;

    cout << "Intersection of A and B:\n";
    set_intersection(A.begin(), A.end(), B.begin(), B.end(), out);
    cout << endl;

    cout << "Difference of A and B:\n";
    set_difference(A.begin(), A.end(), B.begin(), B.end(), out);
    cout << endl;

    set<string> C;
    cout << "Set C:\n";
    set_union(A.begin(), A.end(), B.begin(), B.end(),
        insert_iterator<set<string> >(C, C.begin()));
    copy(C.begin(), C.end(), out);
    cout << endl;

    string s3("grungy");
    C.insert(s3);
    cout << "Set C after insertion:\n";
    copy(C.begin(), C.end(),out);
    cout << endl;

    cout << "Showing a range:\n";
    copy(C.lower_bound("ghost"),C.upper_bound("spook"), out);
    cout << endl;

    return 0;
}
```

下面是程序清单 16.13 中程序的输出：

```
Set A: buffoon can for heavy thinkers
Set B: any deliver elegant food for metal
Union of A and B:
any buffoon can deliver elegant food for heavy metal thinkers
Intersection of A and B:
for
Difference of A and B:
buffoon can heavy thinkers
Set C:
any buffoon can deliver elegant food for heavy metal thinkers
Set C after insertion:
any buffoon can deliver elegant food for grungy heavy metal thinkers
Showing a range:
grungy heavy metal
```

和本章中大多数示例一样，程序清单 16.13 在处理名称空间 std 时采取了偷懒的方式：

```
using namespace std;
```

这样做旨在简化表示方式。这些示例使用了名称空间 std 中非常多的元素，如果使用 using 声明或作用域运算符，代码将变得混乱：

```
std::set<std::string> B(s2, s2 + N);
std::ostream_iterator<std::string, char> out(std::cout, " ");
```

```
std::cout << "Set A: ";
std::copy(A.begin(), A.end(), out);
```

2. multimap 示例

与 set 相似，multimap 也是可反转的、经过排序的关联容器，但键和值的类型不同，且同一个键可能与多个值相关联。

基本的 multimap 声明使用模板参数指定键的类型和存储的值类型。例如，下面的声明创建一个 multimap 对象，其中键类型为 int，存储的值类型为 string：

```
multimap<int,string> codes;
```

第 3 个模板参数是可选的，指出用于对键进行排序的比较函数或对象。在默认情况下，将使用模板 less<>（稍后将讨论），该模板将键类型作为参数。老式 C++实现可能要求显式指定该模板参数。

为将信息结合在一起，实际的值类型将键类型和数据类型结合为一对。为此，STL 使用模板类 pair<class T, class U>将这两种值存储到一个对象中。如果 keytype 是键类型，而 datatype 是存储的数据类型，则值类型为 pair<const keytype, datatype>。例如，前面声明的 codes 对象的值类型为 pair<const int, string>。

例如，假设要用区号作为键来存储城市名（这恰好与 codes 声明一致，它将键类型声明为 int，数据类型声明为 string），则一种方法是创建一个 pair，再将它插入：

```
pair<const int, string> item(213, "Los Angeles");
codes.insert(item);
```

也可使用一条语句创建匿名 pair 对象并将它插入：

```
codes.insert(pair<const int, string> (213, "Los Angeles"));
```

因为数据项是按键排序的，所以不需要指出插入位置。

对于 pair 对象，可以使用 first 和 second 成员来访问其两个部分了：

```
pair<const int, string> item(213, "Los Angeles");
cout << item.first << ' ' << item.second << endl;
```

如何获得有关 multimap 对象的信息呢？成员函数 count( )接受键作为参数，并返回具有该键的元素数目。成员函数 lower_bound( )和 upper_bound( )将键作为参数，且工作原理与处理 set 时相同。成员函数 equal_range( )用键作为参数，且返回两个迭代器，它们表示的区间与该键匹配。为返回两个值，该方法将它们封装在一个 pair 对象中，这里 pair 的两个模板参数都是迭代器。例如，下面的代码打印 codes 对象中区号为 718 的所有城市：

```
pair<multimap<KeyType, string>::iterator,
     multimap<KeyType, string>::iterator> range
                          = codes.equal_range(718);
cout << "Cities with area code 718:\n";
std::multimap<KeyType, std::string>::iterator it;
for (it = range.first; it != range.second; ++it)
    cout <<  (*it).second    << endl;
```

在声明中可使用 C++11 自动类型推断功能，这样代码将简化为如下所示：

```
auto range = codes.equal_range(718);
cout << "Cities with area code 718:\n";
for (auto it = range.first; it != range.second; ++it)
    cout <<  (*it).second    << endl;
```

程序清单 16.14 演示了上述大部分技术，它也使用 typedef 来简化代码：

程序清单 16.14　multimap.cpp

```
// multmap.cpp -- use a multimap
#include <iostream>
#include <string>
```

```
#include <map>
#include <algorithm>

typedef int KeyType;
typedef std::pair<const KeyType, std::string> Pair;
typedef std::multimap<KeyType, std::string> MapCode;

int main()
{
    using namespace std;
    MapCode codes;

    codes.insert(Pair(415, "San Francisco"));
    codes.insert(Pair(510, "Oakland"));
    codes.insert(Pair(718, "Brooklyn"));
    codes.insert(Pair(718, "Staten Island"));
    codes.insert(Pair(415, "San Rafael"));
    codes.insert(Pair(510, "Berkeley"));

    cout << "Number of cities with area code 415: "
         << codes.count(415) << endl;
    cout << "Number of cities with area code 718: "
         << codes.count(718) << endl;
    cout << "Number of cities with area code 510: "
         << codes.count(510) << endl;
    cout << "Area Code   City\n";
    MapCode::iterator it;
    for (it = codes.begin(); it != codes.end(); ++it)
        cout << "    " << (*it).first << "     "
            << (*it).second    << endl;

    pair<MapCode::iterator, MapCode::iterator> range
         = codes.equal_range(718);
    cout << "Cities with area code 718:\n";
    for (it = range.first; it != range.second; ++it)
        cout <<  (*it).second    << endl;

    return 0;
}
```

下面是程序清单 16.14 中程序的输出：

```
Number of cities with area code 415: 2
Number of cities with area code 718: 2
Number of cities with area code 510: 2
Area Code   City
    415     San Francisco
    415     San Rafael
    510     Oakland
    510     Berkeley
    718     Brooklyn
    718     Staten Island
Cities with area code 718:
Brooklyn
Staten Island
```

16.4.5　无序关联容器（C++11）

无序关联容器是对容器概念的另一种改进。与关联容器一样，无序关联容器也将值与键关联起来，并使用键来查找值。但底层的差别在于，关联容器是基于树结构的，而无序关联容器是基于数据结构哈希表的，这旨在提高添加和删除元素的速度以及提高查找算法的效率。有 4 种无序关联容器，它们是 unordered_set、unordered_multiset、unordered_map 和 unordered_multimap，将在附录 G 更详细地介绍。

16.5　函数对象

很多 STL 算法都使用函数对象——也叫函数符（functor）。函数符是可以以函数方式与( )结合使用的任意对象。这包括函数名、指向函数的指针和重载了( )运算符的类对象（即定义了函数 operator( )( )的类）。例如，可以像这样定义一个类：

```
class Linear
{
private:
    double slope;
    double y0;
public:
    Linear(double sl_ = 1, double y_ = 0)
        : slope(sl_), y0(y_) {}
    double operator()(double x) {return y0 + slope * x; }
};
```

这样，重载的( )运算符将使得能够像函数那样使用 Linear 对象：

```
Linear f1;
Linear f2(2.5, 10.0);
double y1 = f1(12.5);   // right-hand side is f1.operator()(12.5)
double y2 = f2(0.4);
```

其中 y1 将使用表达式 0 + 1 * 12.5 来计算，y2 将使用表达式 10.0 + 2.5 * 0.4 来计算。在表达式 y0 + slope * x 中，y0 和 slope 的值来自对象的构造函数，而 x 的值来自 operator( ) ( )的参数。

还记得函数 for_each 吗？它将指定的函数用于区间中的每个成员：

```
for_each(books.begin(), books.end(), ShowReview);
```

通常，第 3 个参数可以是常规函数，也可以是函数符。实际上，这提出了一个问题：如何声明第 3 个参数呢？不能把它声明为函数指针，因为函数指针指定了参数类型。由于容器可以包含任意类型，所以预先无法知道应使用哪种参数类型。STL 通过使用模板解决了这个问题。for_each 的原型看上去就像这样：

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);
```

ShowReview( )的原型如下：

```
void ShowReview(const Review &);
```

这样，标识符 ShowReview 的类型将为 void(*)(const Review &)，这也是赋给模板参数 Function 的类型。对于不同的函数调用，Function 参数可以表示具有重载的( )运算符的类类型。最终，for_each( )代码将具有一个使用 f( )的表达式。在 ShowReview( )示例中，f 是指向函数的指针，而 f( )调用该函数。如果最后的 for_each( )参数是一个对象，则 f( )将是调用其重载的( )运算符的对象。

16.5.1　函数符概念

正如 STL 定义了容器和迭代器的概念一样，它也定义了函数符概念。

- 生成器（generator）是不用参数就可以调用的函数符。
- 一元函数（unary function）是用一个参数可以调用的函数符。
- 二元函数（binary function）是用两个参数可以调用的函数符。

例如，提供给 for_each( )的函数符应当是一元函数，因为它每次用于一个容器元素。

当然，这些概念都有相应的改进版：

- 返回 bool 值的一元函数是谓词（predicate）；
- 返回 bool 值的二元函数是二元谓词（binary predicate）。

一些 STL 函数需要谓词参数或二元谓词参数。例如，程序清单 16.9 使用了 sort( )的这样一个版本，即将二元谓词作为其第 3 个参数：

```
bool WorseThan(const Review & r1, const Review & r2);
...
sort(books.begin(), books.end(), WorseThan);
```

list 模板有一个将谓词作为参数的 remove_if( )成员，该函数将谓词应用于区间中的每个元素，如果谓词返回 true，则删除这些元素。例如，下面的代码删除链表 three 中所有大于 100 的元素：

```
bool tooBig(int n){ return n > 100; }
list<int> scores;
...
scores.remove_if(tooBig);
```

最后这个例子演示了类函数符适用的地方。假设要删除另一个链表中所有大于 200 的值。如果能将取舍值作为第二个参数传递给 tooBig( )，则可以使用不同的值调用该函数，但谓词只能有一个参数。然而，如果设计一个 TooBig 类，则可以使用类成员而不是函数参数来传递额外的信息：

```
template<class T>
class TooBig
{
private:
    T cutoff;
public:
    TooBig(const T & t) : cutoff(t) {}
    bool operator()(const T & v) { return v > cutoff; }
};
```

这里，一个值（V）作为函数参数传递，而第二个参数（cutoff）是由类构造函数设置的。有了该定义后，就可以将不同的 TooBig 对象初始化为不同的取舍值，供调用 remove_if( )时使用。程序清单 16.15 演示了这种技术。

程序清单 16.15　functor.cpp

```
// functor.cpp -- using a functor
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>

template<class T>  // functor class defines operator()()
class TooBig
{
private:
    T cutoff;
public:
    TooBig(const T & t) : cutoff(t) {}
    bool operator()(const T & v) { return v > cutoff; }
```

```
};

void outint(int n) {std::cout << n << " ";}
int main()
{
    using std::list;
    using std::cout;
    using std::endl;

    TooBig<int> f100(100); // limit = 100
    int vals[10] = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
    list<int> yadayada(vals, vals + 10); // range constructor
    list<int> etcetera(vals, vals + 10);
 // C++11 can use the following instead
//  list<int> yadayada = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
//  list<int> etcetera {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
    cout << "Original lists:\n";
    for_each(yadayada.begin(), yadayada.end(), outint);
    cout << endl;
    for_each(etcetera.begin(), etcetera.end(), outint);
    cout << endl;
    yadayada.remove_if(f100);                // use a named function object
    etcetera.remove_if(TooBig<int>(200));    // construct a function object
    cout <<"Trimmed lists:\n";
    for_each(yadayada.begin(), yadayada.end(), outint);
    cout << endl;
    for_each(etcetera.begin(), etcetera.end(), outint);
    cout << endl;
    return 0;
}
```

一个函数符（f100）是一个声明的对象，而另一个函数符（TooBig<int>(200)）是一个匿名对象，它是由构造函数调用创建的。下面是程序清单 16.15 中程序的输出：

```
Original lists:
50 100 90 180 60 210 415 88 188 201
50 100 90 180 60 210 415 88 188 201
Trimmed lists:
50 100 90 60 88
50 100 90 180 60 88 188
```

假设已经有了一个接受两个参数的模板函数：

```
template <class T>
bool tooBig(const T & val, const T & lim)
{
    return val > lim;
}
```

则可以使用类将它转换为单个参数的函数对象：

```
template<class T>
class TooBig2
{
private:
    T cutoff;
public:
    TooBig2(const T & t) : cutoff(t) {}
```

```
    bool operator()(const T & v) { return tooBig<T>(v, cutoff); }
};
```

即可以这样做：

```
TooBig2<int> tB100(100);
int x;
cin >> x;
if (tB100(x))   // same as if (tooBig(x,100))
    ...
```

因此，调用 tB100(x)相当于调用 tooBig(x, 100)，但两个参数的函数被转换为单参数的函数对象，其中第二个参数被用于构建函数对象。简而言之，类函数符 TooBig2 是一个函数适配器，使函数能够满足不同的接口。

在该程序清单中，可使用 C++11 的初始化列表功能来简化初始化。为此，可将如下代码：

```
int vals[10] = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
list<int> yadayada(vals, vals + 10); // range constructor
list<int> etcetera(vals, vals + 10);
```

替换为下述代码：

```
list<int> yadayada = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
list<int> etcetera {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
```

16.5.2 预定义的函数符

STL 定义了多个基本函数符，它们执行诸如将两个值相加、比较两个值是否相等操作。提供这些函数对象是为了支持将函数作为参数的 STL 函数。例如，考虑函数 transform( )。它有两个版本。第一个版本接受 4 个参数，前两个参数是指定容器区间的迭代器（现在您应该已熟悉了这种方法），第 3 个参数是指定将结果复制到哪里的迭代器，最后一个参数是一个函数符，它被应用于区间中的每个元素，生成结果中的新元素。例如，请看下面的代码：

```
const int LIM = 5;
double arr1[LIM] = {36, 39, 42, 45, 48};
vector<double> gr8(arr1, arr1 + LIM);
ostream_iterator<double, char> out(cout, " ");
transform(gr8.begin(), gr8.end(), out, sqrt);
```

上述代码计算每个元素的平方根，并将结果发送到输出流。目标迭代器可以位于原始区间中。例如，将上述示例中的 out 替换为 gr8.begin( )后，新值将覆盖原来的值。很明显，使用的函数符必须是接受单个参数的函数符。

第 2 种版本使用一个接受两个参数的函数，并将该函数用于两个区间中元素。它用另一个参数（即第 3 个）标识第二个区间的起始位置。例如，如果 m8 是另一个 vector<double>对象，mean（double，double）返回两个值的平均值，则下面的的代码将输出来自 gr8 和 m8 的值的平均值：

```
transform(gr8.begin(), gr8.end(), m8.begin(), out, mean);
```

现在假设要将两个数组相加。不能将+作为参数，因为对于类型 double 来说，+是内置的运算符，而不是函数。可以定义一个将两个数相加的函数，然后使用它：

```
double add(double x, double y) { return x + y; }
...
transform(gr8.begin(), gr8.end(), m8.begin(), out, add);
```

然而，这样必须为每种类型单独定义一个函数。更好的办法是定义一个模板（除非 STL 已经有一个模板了，这样就不必定义）。头文件 functional（以前为 function.h）定义了多个模板类函数对象，其中包括 plus< >( )。

可以用 plus< >类来完成常规的相加运算：

```
#include <functional>
...
plus<double> add;  // create a plus<double> object
double y = add(2.2, 3.4); // using plus<double>::operator()()
```

它使得将函数对象作为参数很方便：

```
transform(gr8.begin(), gr8.end(), m8.begin(), out, plus<double>() );
```

这里，代码没有创建命名的对象，而是用 plus<double>构造函数构造了一个函数符，以完成相加运算（括号表示调用默认的构造函数，传递给 transform( )的是构造出来的函数对象）。

对于所有内置的算术运算符、关系运算符和逻辑运算符，STL 都提供了等价的函数符。表 16.12 列出了这些函数符的名称。它们可以用于处理 C++内置类型或任何用户定义类型（如果重载了相应的运算符）。

表 16.12　运算符和相应的函数符

运　算　符	相应的函数符
+	plus
-	minus
*	multiplies
/	divides
%	modulus
-	negate
==	equal_to
!=	not_equal_to
>	greater
<	less
>=	greater_equal
<=	less_equal
&&	logical_and
||	logical_or
!	logical_not

警告：老式 C++实现使用函数符名 times，而不是 multiplies。

16.5.3　自适应函数符和函数适配器

表 16.12 列出的预定义函数符都是自适应的。实际上 STL 有 5 个相关的概念：自适应生成器（adaptable generator）、自适应一元函数（adaptable unary function）、自适应二元函数（adaptable binary function）、自适应谓词（adaptable predicate）和自适应二元谓词（adaptable binary predicate）。

使函数符成为自适应的原因是，它携带了标识参数类型和返回类型的 typedef 成员。这些成员分别是 result_type、first_argument_type 和 second_argument_type，它们的作用是不言自明的。例如，plus<int>对象的返回类型被标识为 plus<int>::result_type，这是 int 的 typedef。

函数符自适应性的意义在于：函数适配器对象可以使用函数对象，并认为存在这些 typedef 成员。例如，接受一个自适应函数符参数的函数可以使用 result_type 成员来声明一个与函数的返回类型匹配的变量。

STL 提供了使用这些工具的函数适配器类。例如，假设要将矢量 gr8 的每个元素都增加 2.5 倍，则需要使用接受一个一元函数参数的 transform( )版本，就像前面的例子那样：

```
transform(gr8.begin(), gr8.end(), out, sqrt);
```

multiplies( )函数符可以执行乘法运行，但它是二元函数。因此需要一个函数适配器，将接受两个参数

的函数符转换为接受 1 个参数的函数符。前面的 TooBig2 示例提供了一种方法，但 STL 使用 binder1st 和 binder2nd 类自动完成这一过程，它们将自适应二元函数转换为自适应一元函数。

来看 binder1st。假设有一个自适应二元函数对象 f2( )，则可以创建一个 binder1st 对象，该对象与一个将被用作 f2( )的第一个参数的特定值（val）相关联：

```
binder1st(f2, val) f1;
```

这样，使用单个参数调用 f1(x)时，返回的值与将 val 作为第一参数、将 f1( )的参数作为第二参数调用 f2( )返回的值相同。即 f1(x)等价于 f2(val, x)，只是前者是一元函数，而不是二元函数。f2( )函数被适配。同样，仅当 f2( )是一个自适应函数时，这才能实现。

看上去有点麻烦。然而，STL 提供了函数 bind1st( )，以简化 binder1st 类的使用。可以问其提供用于构建 binder1st 对象的函数名称和值，它将返回一个这种类型的对象。例如，要将二元函数 multiplies( )转换为将参数乘以 2.5 的一元函数，则可以这样做：

```
bind1st(multiplies<double>(), 2.5)
```

因此，将 gr8 中的每个元素与 2.5 相乘，并显示结果的代码如下：

```
transform(gr8.begin(), gr8.end(), out,
        bind1st(multiplies<double>(), 2.5));
```

binder2nd 类与此类似，只是将常数赋给第二个参数，而不是第一个参数。它有一个名为 bind2nd 的助手函数，该函数的工作方式类似于 bind1st。

程序清单 16.16 将一些最近的示例合并成了一个小程序。

程序清单 16.16　funadap.cpp

```
// funadap.cpp -- using function adapters
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
#include <functional>

void Show(double);
const int LIM = 6;
int main()
{
    using namespace std;
    double arr1[LIM] = {28, 29, 30, 35, 38, 59};
    double arr2[LIM] = {63, 65, 69, 75, 80, 99};
    vector<double> gr8(arr1, arr1 + LIM);
    vector<double> m8(arr2, arr2 + LIM);
    cout.setf(ios_base::fixed);
    cout.precision(1);
    cout << "gr8:\t";
    for_each(gr8.begin(), gr8.end(), Show);
    cout << endl;
    cout << "m8: \t";
    for_each(m8.begin(), m8.end(), Show);
    cout << endl;

    vector<double> sum(LIM);
    transform(gr8.begin(), gr8.end(), m8.begin(), sum.begin(),
              plus<double>());
    cout << "sum:\t";
    for_each(sum.begin(), sum.end(), Show);
```

```
    cout << endl;

    vector<double> prod(LIM);
    transform(gr8.begin(), gr8.end(), prod.begin(),
              bind1st(multiplies<double>(), 2.5));
    cout << "prod:\t";
    for_each(prod.begin(), prod.end(), Show);
    cout << endl;

    return 0;
}
void Show(double v)
{
    std::cout.width(6);
    std::cout << v << ' ';
}
```

程序清单 16.16 中程序的输出如下：

```
gr8:      28.0   29.0   30.0   35.0   38.0   59.0
m8:       63.0   65.0   69.0   75.0   80.0   99.0
sum:      91.0   94.0   99.0  110.0  118.0  158.0
prod:     70.0   72.5   75.0   87.5   95.0  147.5
```

C++11 提供了函数指针和函数符的替代品——lambda 表达式，这将在第 18 章讨论。

16.6　算法

STL 包含很多处理容器的非成员函数，前面已经介绍过其中的一些：sort( )、copy( )、find( )、random_shuffle( )、set_union( )、set_intersection( )、set_difference( )和 transform( )。可能已经注意到，它们的总体设计是相同的，都使用迭代器来标识要处理的数据区间和结果的放置位置。有些函数还接受一个函数对象参数，并使用它来处理数据。

对于算法函数设计，有两个主要的通用部分。首先，它们都使用模板来提供泛型；其次，它们都使用迭代器来提供访问容器中数据的通用表示。因此，copy( )函数可用于将 double 值存储在数组中的容器、将 string 值存储在链表中的容器，也可用于将用户定义的对象存储在树结构中（如 set 所使用的）的容器。因为指针是一种特殊的迭代器，因此诸如 copy( )等 STL 函数可用于常规数组。

统一的容器设计使得不同类型的容器之间具有明显关系。例如，可以使用 copy( )将常规数组中的值复制到 vector 对象中，将 vector 对象中的值复制到 list 对象中，将 list 对象中的值复制到 set 对象中。可以用= =来比较不同类型的容器，如 deque 和 vector。之所以能够这样做，是因为容器重载的= =运算符使用迭代器来比较内容，因此如果 deque 对象和 vector 对象的内容相同，并且排列顺序也相同，则它们是相等的。

16.6.1　算法组

STL 将算法库分成 4 组：

- 非修改式序列操作；
- 修改式序列操作；
- 排序和相关操作；
- 通用数字运算。

前 3 组在头文件 algorithm（以前为 algo.h）中描述，第 4 组是专用于数值数据的，有自己的头文件，称为 numeric（以前它们也位于 algol.h 中）。

非修改式序列操作对区间中的每个元素进行操作。这些操作不修改容器的内容。例如，find( )和for_each( )就属于这一类。

修改式序列操作也对区间中的每个元素进行操作。然而，顾名思义，它们可以修改容器的内容。可以修改值，也可以修改值的排列顺序。transform( )、random_shuffle( )和 copy( )属于这一类。

排序和相关操作包括多个排序函数（包括 sort( )）和其他各种函数，包括集合操作。

数字操作包括将区间的内容累积、计算两个容器的内部乘积、计算小计、计算相邻对象差的函数。通常，这些都是数组的操作特性，因此 vector 是最有可能使用这些操作的容器。

16.6.2 算法的通用特征

正如您多次看到的，STL 函数使用迭代器和迭代器区间。从函数原型可知有关迭代器的假设。例如，copy( )函数的原型如下：

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

因为标识符 InputIterator 和 OutputIterator 都是模板参数，所以它们就像 T 和 U 一样。然而，STL 文档使用模板参数名称来表示参数模型的概念。因此上述声明告诉我们，区间参数必须是输入迭代器或更高级别的迭代器，而指示结果存储位置的迭代器必须是输出迭代器或更高级别的迭代器。

对算法进行分类的方式之一是按结果放置的位置进行分类。有些算法就地完成工作，有些则创建拷贝。例如，在 sort( )函数完成时，结果被存放在原始数据的位置上，因此，sort( )是就地算法（in-place algorithm）；而 copy( )函数将结果发送到另一个位置，所以它是复制算法（copying algorithm）。transform( )函数可以以这两种方式完成工作。与 copy( )相似，它使用输出迭代器指示结果的存储位置；与 copy( )不同的是，transform( )允许输出迭代器指向输入区间，因此它可以用计算结果覆盖原来的值。

有些算法有两个版本：就地版本和复制版本。STL 的约定是，复制版本的名称将以_copy 结尾。复制版本将接受一个额外的输出迭代器参数，该参数指定结果的放置位置。例如，函数 replace( )的原型如下：

```
template<class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value);
```

它将所有的 old_value 替换为 new_value，这是就地发生的。由于这种算法同时读写容器元素，因此迭代器类型必须是 ForwardIterator 或更高级别的。复制版本的原型如下：

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
             OutputIterator result,
             const T& old_value, const T& new_value);
```

在这里，结果被复制到 result 指定的新位置，因此对于指定区间而言，只读输入迭代器足够了。

注意，replace_copy( )的返回类型为 OutputIterator。对于复制算法，统一的约定是：返回一个迭代器，该迭代器指向复制的最后一个值后面的一个位置。

另一个常见的变体是：有些函数有这样的版本，即根据将函数应用于容器元素得到的结果来执行操作。这些版本的名称通常以_if 结尾。例如，如果将函数用于旧值时，返回的值为 true，则 replace_if( )将把旧值替换为新的值。下面是该函数的原型：

```
template<class ForwardIterator, class Predicate class T>
void replace_if(ForwardIterator first, ForwardIterator last,
             Predicate pred, const T& new_value);
```

如前所述，谓词是返回 bool 值的一元函数。还有一个 replace_copy_if( )版本，您不难知道其作用和原型。

与 InputIterator 一样，Predicate 也是模板参数名称，可以为 T 或 U。然而，STL 选择用 Predicate 来提

醒用户，实参应模拟 Predicate 概念。同样，STL 使用诸如 Generator 和 BinaryPredicate 等术语来指示必须模拟其他函数对象概念的参数。请记住，虽然文档可指出迭代器或函数符需求，但编译器不会对此进行检查。如果您使用了错误的迭代器，则编译器试图实例化模板时，将显示大量的错误消息。

16.6.3　STL 和 string 类

string 类虽然不是 STL 的组成部分，但设计它时考虑到了 STL。例如，它包含 begin( )、end( )、rbegin( ) 和 rend( )等成员，因此可以使用 STL 接口。程序清单 16.17 用 STL 显示了使用一个词的字母可以得到的所有排列组合。排列组合就是重新安排容器中元素的顺序。next_permutation( )算法将区间内容转换为下一种排列方式。对于字符串，排列按照字母递增的顺序进行。如果成功，该算法返回 true；如果区间已经处于最后的序列中，则该算法返回 false。要得到区间内容的所有排列组合，应从最初的顺序开始，为此程序使用了 STL 算法 sort( )。

程序清单 16.17　strgst1.cpp

```
// strgstl.cpp -- applying the STL to a string
#include <iostream>
#include <string>
#include <algorithm>

int main()
{
    using namespace std;
    string letters;
    cout << "Enter the letter grouping (quit to quit): ";
    while (cin >> letters && letters != "quit")
    {
        cout << "Permutations of " << letters << endl;
        sort(letters.begin(), letters.end());
        cout << letters << endl;
        while (next_permutation(letters.begin(), letters.end()))
            cout << letters << endl;
        cout << "Enter next sequence (quit to quit): ";
    }
    cout << "Done.\n";
    return 0;
}
```

程序清单 16.17 中程序的运行情况如下：

```
Enter the letter grouping (quit to quit): awl
Permutations of awl
alw
awl
law
lwa
wal
wla
Enter next sequence (quit to quit): all
Permutations of all
all
lal
lla
Enter next sequence (quit to quit): quit
Done.
```

注意，算法 next_permutation( )自动提供唯一的排列组合，这就是输出中“awl”一词的排列组合比“all”（它有重复的字母）的排列组合要多的原因。

16.6.4 函数和容器方法

有时可以选择使用 STL 方法或 STL 函数。通常方法是更好的选择。首先，它更适合于特定的容器；其次，作为成员函数，它可以使用模板类的内存管理工具，从而在需要时调整容器的长度。

例如，假设有一个由数字组成的链表，并要删除链表中某个特定值（例如 4）的所有实例。如果 la 是一个 list<int>对象，则可以使用链表的 remove( )方法：

```
la.remove(4);  // remove all 4s from the list
```

调用该方法后，链表中所有值为 4 的元素都将被删除，同时链表的长度将被自动调整。

还有一个名为 remove( )的 STL 算法（见附录 G），它不是由对象调用，而是接受区间参数。因此，如果 lb 是一个 list<int>对象，则调用该函数的代码如下：

```
remove(lb.begin(), lb.end(), 4);
```

然而，由于该 remove( )函数不是成员，因此不能调整链表的长度。它将没被删除的元素放在链表的开始位置，并返回一个指向新的超尾值的迭代器。这样，便可以用该迭代器来修改容器的长度。例如，可以使用链表的 erase( )方法来删除一个区间，该区间描述了链表中不再需要的部分。程序清单 16.18 演示了这是如何进行的。

程序清单 16.18　listrmv.cpp

```
// listrmv.cpp -- applying the STL to a string
#include <iostream>
#include <list>
#include <algorithm>
void Show(int);
const int LIM = 10;
int main()
{
    using namespace std;
    int ar[LIM] = {4, 5, 4, 2, 2, 3, 4, 8, 1, 4};
    list<int> la(ar, ar + LIM);
    list<int> lb(la);
    cout << "Original list contents:\n\t";
    for_each(la.begin(), la.end(), Show);
    cout << endl;
    la.remove(4);
    cout << "After using the remove() method:\n";
    cout << "la:\t";
    for_each(la.begin(), la.end(), Show);
    cout << endl;
    list<int>::iterator last;
    last = remove(lb.begin(), lb.end(), 4);
    cout << "After using the remove() function:\n";
    cout << "lb:\t";
    for_each(lb.begin(), lb.end(), Show);
    cout << endl;
    lb.erase(last, lb.end());
    cout << "After using the erase() method:\n";
    cout << "lb:\t";
    for_each(lb.begin(), lb.end(), Show);
    cout << endl;
```

```
    return 0;
}

void Show(int v)
{
    std::cout << v << ' ';
}
```

下面是程序清单 16.18 中程序的输出：

```
Original list contents:
    4 5 4 2 2 3 4 8 1 4
After using the remove() method:
la: 5 2 2 3 8 1
After using the remove() function:
lb: 5 2 2 3 8 1 4 8 1 4
After using the erase() method:
lb: 5 2 2 3 8 1
```

从中可知，remove( )方法将链表 la 从 10 个元素减少到 6 个元素。但对链表 lb 应用 remove( )后，它仍然包含 10 个元素。最后 4 个元素可任意处理，因为其中每个元素要么为 4，要么与已经移到链表开头的值相同。

尽管方法通常更适合，但非方法函数更通用。正如您看到的，可以将它们用于数组、string 对象、STL 容器，还可以用它们来处理混合的容器类型，例如，将矢量容器中的数据存储到链表或集合中。

16.6.5　使用 STL

STL 是一个库，其组成部分被设计成协同工作。STL 组件是工具，但也是创建其他工具的基本部件。我们用一个例子说明。假设要编写一个程序，让用户输入单词。希望最后得到一个按输入顺序排列的单词列表、一个按字母顺序排列的单词列表（忽略大小写），并记录每个单词被输入的次数。出于简化的目的，假设输入中不包含数字和标点符号。

输入和保存单词列表很简单。可以按程序清单 16.8 和程序清单 16.9 那样创建一个 vector<string>对象，并用 push_back( )将输入的单词添加到矢量中：

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

如何得到按字母顺序排列的单词列表呢？可以使用 sort( )，然后使用 unique( )，但这种方法将覆盖原始数据，因为 sort( )是就地算法。有一种更简单的方法，可以避免这种问题：创建一个 set<string>对象，然后将矢量中的单词复制（使用插入迭代器）到集合中。集合自动对其内容进行排序，因此无需调用 sort( )；集合只允许同一个键出现一次，因此无需调用 unique( )。这里要求忽略大小写，处理这种情况的方法之一是使用 transform( )而不是 copy( )，将矢量中的数据复制到集合中。使用一个转换函数将字符串转换成小写形式。

```
set<string> wordset;
transform(words.begin(), words.end(),
    insert_iterator<set<string> > (wordset, wordset.begin()), ToLower);
```

ToLower( )函数很容易编写，只需使用 transform( )将 tolower( )函数应用于字符串中的各个元素，并将字符串用作源和目标。记住，string 对象也可以使用 STL 函数。将字符串按引用传递和返回意味着算法不必复制字符串，而可以直接操作原始字符串。下面是函数 ToLower( )的代码：

```
string & ToLower(string & st)
{
    transform(st.begin(), st.end(), st.begin(), tolower);
    return st;
}
```

一个可能出现的问题是：tolower( )函数被定义为 int tolower（int），而一些编译器希望函数与元素类型（即 char）匹配。一种解决方法是，使用 toLower 代替 tolower，并提供下面的定义：

```
char toLower(char ch) { return tolower(ch); }
```

要获得每个单词在输入中出现的次数，可以使用 count( )函数。它将一个区间和一个值作为参数，并返回这个值在区间中出现的次数。可以使用 vector 对象来提供区间，并使用 set 对象来提供要计算其出现次数的单词列表。即对于集合中的每个词，都计算它在矢量中出现的次数。要将单词与其出现的次数关联起来，可将单词和计数作为 pair<const string, int>对象存储在 map 对象中。单词将作为键（只出现一次），计数作为值。这可以通过一个循环来完成：

```
map<string, int> wordmap;
set<string>::iterator  si;
for (si = wordset.begin(); si != wordset.end(); si++)
    wordmap.insert(pair<string, int>(*si, count(words.begin(),
    words.end(), *si)));
```

map 类有一个有趣的特征：可以用数组表示法（将键用作索引）来访问存储的值。例如，wordmap[“the”]表示与键“the”相关联的值，这里是字符串“the”出现的次数。因为 wordset 容器保存了 wordmap 使用的全部键，所以可以用下面的代码来存储结果，这是一种更具吸引力的方法：

```
for (si = wordset.begin(); si != wordset.end(); si++)
    wordmap[*si] = count(words.begin(), words.end(), *si);
```

因为 si 指向 wordset 容器中的一个字符串，所以*si 是一个字符串，可以用作 wordmap 的键。上述代码将键和值都放到 wordmap 映象中。

同样，也可以使用数组表示法来报告结果：

```
for (si = wordset.begin(); si != wordset.end(); si++)
    cout << *si << ": " << wordmap[*si] << endl;
```

如果键无效，则对应的值将为 0。

程序清单 16.19 把这些想法组合在一起，同时包含了用于显示 3 个容器（包含输入内容的矢量、包含单词列表的集合和包含单词计数的映象）内容的代码。

程序清单 16.19　usealgo.cpp

```
//usealgo.cpp -- using several STL elements
#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <map>
#include <iterator>
#include <algorithm>
#include <cctype>
using namespace std;

char toLower(char ch) { return tolower(ch); }
string & ToLower(string & st);
void display(const string & s);

int main()
{
    vector<string> words;
    cout << "Enter words (enter quit to quit):\n";
    string input;
    while (cin >> input && input != "quit")
```

```
    words.push_back(input);

cout << "You entered the following words:\n";
for_each(words.begin(), words.end(), display);
cout << endl;

// place words in set, converting to lowercase
set<string> wordset;
transform(words.begin(), words.end(),
    insert_iterator<set<string> > (wordset, wordset.begin()),
    ToLower);
cout << "\nAlphabetic list of words:\n";
for_each(wordset.begin(), wordset.end(), display);
cout << endl;

// place word and frequency in map
map<string, int> wordmap;
set<string>::iterator si;
for (si = wordset.begin(); si != wordset.end(); si++)
    wordmap[*si] = count(words.begin(), words.end(), *si);

// display map contents
cout << "\nWord frequency:\n";
for (si = wordset.begin(); si != wordset.end(); si++)
    cout << *si << ": " << wordmap[*si] << endl;

return 0;
}

string & ToLower(string & st)
{
    transform(st.begin(), st.end(), st.begin(), toLower);
    return st;
}

void display(const string & s)
{
    cout << s << " ";
}
```

程序清单 16.19 中程序的运行情况如下：

```
Enter words (enter quit to quit):
The dog saw the cat and thought the cat fat
The cat thought the cat perfect
quit
You entered the following words:
The dog saw the cat and thought the cat fat The cat thought the cat perfect

Alphabetic list of words:
and cat dog fat perfect saw the thought

Word frequency:
and: 1
cat: 4
dog: 1
```

```
fat: 1
perfect: 1
saw: 1
the: 5
thought: 2
```

这里的寓意在于，使用 STL 时应尽可能减少要编写的代码。STL 通用、灵活的设计将节省大量工作。另外，STL 设计者就是非常关心效率的算法人员，算法是经过仔细选择的，并且是内联的。

16.7 其他库

C++还提供了其他一些类库，它们比本章讨论前面的例子更为专用。例如，头文件 complex 为复数提供了类模板 complex，包含用于 float、long 和 long double 的具体化。这个类提供了标准的复数运算及能够处理复数的标准函数。C++11 新增的头文件 random 提供了更多的随机数功能。

第 14 章介绍了头文件 valarray 提供的模板类 valarray。这个类模板被设计成用于表示数值数组，支持各种数值数组操作，例如将两个数组的内容相加、对数组的每个元素应用数学函数以及对数组进行线性代数运算。

16.7.1 vector、valarray 和 array

您可能会问，C++为何提供三个数组模板：vector、valarray 和 array。这些类是由不同的小组开发的，用于不同的目的。vector 模板类是一个容器类和算法系统的一部分，它支持面向容器的操作，如排序、插入、重新排列、搜索、将数据转移到其他容器中等。而 valarray 类模板是面向数值计算的，不是 STL 的一部分。例如，它没有 push_back( )和 insert( )方法，但为很多数学运算提供了一个简单、直观的接口。最后，array 是为替代内置数组而设计的，它通过提供更好、更安全的接口，让数组更紧凑，效率更高。Array 表示长度固定的数组，因此不支持 push_back( )和 insert( )，但提供了多个 STL 方法，包括 begin( )、end( )、rbegin( )和 rend( )，这使得很容易将 STL 算法用于 array 对象。

例如，假设有如下声明：

```
vector<double> ved1(10), ved2(10), ved3(10);
array<double, 10> vod1, vod2, vod3;
valarray<double> vad1(10), vad2(10), vad3(10);
```

同时，假设 ved1、ved2、vod1、vod2、vad1 和 vad2 都有合适的值。要将两个数组中第一个元素的和赋给第三个数组的第一个元素，使用 vector 类时，可以这样做：

```
transform(ved1.begin(), ved1.end(), ved2.begin(), ved3.begin(),
          plus<double>());
```

对于 array 类，也可以这样做：

```
transform(vod1.begin(), vod1.end(), vod2.begin(), vod3.begin(),
          plus<double>());
```

然而，valarray 类重载了所有算术运算符，使其能够用于 valarray 对象，因此您可以这样做：

```
vad3 = vad1 + vad2;    // + overloaded
```

同样，下面的语句将使 vad3 中每个元素都是 vad1 和 vad2 中相应元素的乘积：

```
vad3 = vad1 * vad2;    // * overloaded
```

要将数组中每个元素的值扩大 2.5 倍，STL 方法如下：

```
transform(ved3.begin(), ved3.end(), ved3.begin(),
          bind1st(multiplies<double>(), 2.5));
```

valarray 类重载了将 valarray 对象乘以一个值的运算符，还重载了各种组合赋值运算符，因此可以采取

下列两种方法之一：

```
vad3 = 2.5 * vad3;      // * overloaded
vad3 *= 2.5;            // *= overloaded
```

假设您要计算数组中每个元素的自然对数，并将计算结果存储到另一个数组的相应元素中，STL 方法如下：

```
transform(ved1.begin(), ved1.end(), ved3.begin(),
          log);
```

valarray 类重载了这种数学函数，使之接受一个 valarray 参数，并返回一个 valarray 对象，因此您可以这样做：

```
vad3 = log(vad1);      // log() overloaded
```

也可以使用 apply( )方法，该方法也适用于非重载函数：

```
vad3 = vad1.apply(log);
```

方法 apply( )不修改调用对象，而是返回一个包含结果的新对象。

执行多步计算时，valarray 接口的简单性将更为明显：

```
vad3 = 10.0* ((vad1 + vad2) / 2.0 + vad1 * cos(vad2));
```

有关使用 STL vector 来完成上述计算的代码留给您去完成。

valarray 类还提供了方法 sum( )（计算 valarray 对象中所有元素的和）、size( )（返回元素数）、max( )（返回最大的元素值）和 min( )（返回最小的元素值）。

正如您看到的，对于数学运算而言，valarray 类提供了比 vector 更清晰的表示方式，但通用性更低。valarray 类确实有一个 resize( )方法，但不能像使用 vector 的 push_back 时那样自动调整大小。没有支持插入、排序、搜索等操作的方法。总之，与 vector 类相比，valarray 类关注的东西更少，但这使得它的接口更简单。

valarray 的接口更简单是否意味着性能更高呢？在大多数情况下，答案是否定的。简单表示法通常是使用类似于您处理常规数组时使用的循环实现的。然而，有些硬件设计允许在执行矢量操作时，同时将一个数组中的值加载到一组寄存器中，然后并行地进行处理。从原则上说，valarray 操作也可以实现成利用这样的设计。

可以将 STL 功能用于 valarray 对象吗？通过回答这个问题，可以快速地复习一些 STL 原理。假设有一个包含 10 个元素的 valarray<double>对象：

```
valarray<double> vad(10);
```

使用数字填充该数组后，能够将 STL sort( )函数用于该数组吗？valarray 类没有 begin( )和 end( )方法，因此不能将它们用作指定区间的参数：

```
sort(vad.begin(), vad.end());  // NO, no begin(), end()
```

另外，vad 是一个对象，而不是指针，因此不能像处理常规数组那样，使用 vad 和 vad + 10 作为区间参数，即下面的代码不可行：

```
sort(vad, vad + 10);  // NO, vad an object, not an address
```

可以使用地址运算符：

```
sort(&vad[0], &vad[10]);  // maybe?
```

但 valarray 没有定义下标超过尾部一个元素的行为。这并不一定意味着使用&vadp[10]不可行。事实上，使用 6 种编译器测试上述代码时，都是可行的；但这确实意味着可能不可行。为让上述代码不可行，需要一个不太可能出现的条件，如让数组与预留给堆的内存块相邻。然而，如果 3.85 亿的交易命悬于您的代码，您可能不想冒代码出现问题的风险。

为解决这种问题，C++11 提供了接受 valarray 对象作为参数的模板函数 begin( )和 end( )。因此，您将使用 begin(vad)而不是 vad.begin。这些函数返回的值满足 STL 区间需求：

```
sort(begin(vad), end(vad)); // C++11 fix!
```

程序清单 16.20 演示了 vector 和 valarray 类各自的优势。它使用 vector 的 push_back( )方法和自动调整大小的功能来收集数据，然后对数字进行排序后，将它们从 vector 对象复制到一个同样大小的 valarray 对象中，再执行一些数学运算。

程序清单 16.20　valvect.cpp

```
// valvect.cpp -- comparing vector and valarray
#include <iostream>
#include <valarray>
#include <vector>
#include <algorithm>
int main()
{
    using namespace std;
    vector<double> data;
    double temp;

    cout << "Enter numbers (<=0 to quit):\n";
    while (cin >> temp && temp > 0)
        data.push_back(temp);
    sort(data.begin(), data.end());
    int size = data.size();
    valarray<double> numbers(size);
    int i;
    for (i = 0; i < size; i++)
        numbers[i] = data[i];
    valarray<double> sq_rts(size);
    sq_rts = sqrt(numbers);
    valarray<double> results(size);
    results = numbers + 2.0 * sq_rts;
    cout.setf(ios_base::fixed);
    cout.precision(4);
    for (i = 0; i < size; i++)
    {
        cout.width(8);
        cout << numbers[i] << ": ";
        cout.width(8);
        cout << results[i] << endl;
    }
    cout << "done\n";
    return 0;
}
```

下面是程序清单 16.20 中程序的运行情况：

```
Enter numbers (<=0 to quit):
3.3 1.8 5.2 10 14.4 21.6 26.9 0
  1.8000:   4.4833
  3.3000:   6.9332
  5.2000:   9.7607
 10.0000:  16.3246
 14.4000:  21.9895
 21.6000:  30.8952
 26.9000:  37.2730
done
```

除前面讨论的外，valarray 类还有很多其他特性。例如，如果 numbers 是一个 valarray<double>对象，则下面的语句将创建一个 bool 数组，其中 vbool[i]被设置为 numbers[i] > 9 的值，即 true 或 false：

```
valarray<bool> vbool = numbers > 9;
```

还有扩展的下标指定版本，来看其中的一个——slice 类。slice 类对象可用作数组索引，在这种情况下，它表的不是一个值而是一组值。slice 对象被初始化为三个整数值：起始索引、索引数和跨距。起始索引是第一个被选中的元素的索引，索引数指出要选择多少个元素，跨距表示元素之间的间隔。例如，slice(1, 4, 3)创建的对象表示选择 4 个元素，它们的索引分别是 1、4、7 和 10。也就是说，从起始索引开始，加上跨距得到下一个元素的索引，依此类推，直到选择了 4 个元素。如果 varint 是一个 valarray<int>对象，则下面的语句将把第 1、4、7、10 个元素都设置为 10：

```
varint[slice(1,4,3)] = 10;  // set selected elements to 10
```

这种特殊的下标指定功能让您能够使用一个一维 valarray 对象来表示二维数据。例如，假设要表示一个 4 行 3 列的数组，可以将信息存储在一个包含 12 个元素的 valarray 对象中，然后使用一个 slice(0, 3, 1)对象作为下标，来表示元素 0、1 和 2，即第 1 行。同样，下标 slice(0, 4, 3)表示元素 0、3、6 和 9，即第一列。程序清单 16.21 演示了 slice 的一些特性。

程序清单 16.21　vslice.cpp

```
// vslice.cpp -- using valarray slices
#include <iostream>
#include <valarray>
#include <cstdlib>

const int SIZE = 12;
typedef std::valarray<int> vint;     // simplify declarations
void show(const vint & v, int cols);
int main()
{
    using std::slice;                // from <valarray>
    using std::cout;
    vint valint(SIZE);               // think of as 4 rows of 3

    int i;
    for (i = 0; i < SIZE; ++i)
        valint[i] = std::rand() % 10;
    cout << "Original array:\n";
    show(valint, 3);                 // show in 3 columns
    vint vcol(valint[slice(1,4,3)]); // extract 2nd column
    cout << "Second column:\n";
    show(vcol, 1);                   // show in 1 column
    vint vrow(valint[slice(3,3,1)]); // extract 2nd row
    cout << "Second row:\n";
    show(vrow, 3);
    valint[slice(2,4,3)]  = 10;      // assign to 2nd column
    cout << "Set last column to 10:\n";
    show(valint, 3);
    cout << "Set first column to sum of next two:\n";
    // + not defined for slices, so convert to valarray<int>
    valint[slice(0,4,3)]  = vint(valint[slice(1,4,3)])
                            + vint(valint[slice(2,4,3)]);
    show(valint, 3);
    return 0;
}
```

```
void show(const vint & v, int cols)
{
    using std::cout;
    using std::endl;

    int lim = v.size();
    for (int i = 0; i < lim; ++i)
    {
        cout.width(3);
        cout << v[i];
        if (i % cols == cols - 1)
            cout << endl;
        else
            cout << ' ';
    }
    if (lim % cols != 0)
        cout << endl;
}
```

对于 valarray 对象（如 valint）和单个 int 元素（如 valint[1]），定义了运算符+；但正如程序清单 16.21 指出的，对于使用 slice 下标指定的 valarray 单元，如 valint[slice(1, 4, 3)，并没有定义运算符+。因此程序使用 slice 指定的元素创建一个完整的 valint 对象，以便能够执行加法运算：

```
vint(valint[slice(1,4,3)])    // calls a slice-based constructor
```

valarray 类提供了用于这种目的的构造函数。

下面是程序清单 16.21 中程序的运行情况：

```
Original array:
  0   3   3
  2   9   0
  8   2   6
  6   9   1
Second column:
  3
  9
  2
  9
Second row:
  2   9   0
Set last column to 10:
  0   3  10
  2   9  10
  8   2  10
  6   9  10
Set first column to sum of next two:
 13   3  10
 19   9  10
 12   2  10
 19   9  10
```

由于元素值是使用 rand( )设置的，因此不同的 rand( )实现将设置不同的值。

另外，使用 gslice 类可以表示多维下标，但上述内容应足以让您对 valarray 有一定了解。

16.7.2 模板 initializer_list（C++11）

模板 initializer_list 是 C++11 新增的。您可使用初始化列表语法将 STL 容器初始化为一系列值：

```
std::vector<double> payments {45.99, 39.23, 19.95, 89.01};
```

这将创建一个包含 4 个元素的容器，并使用列表中的 4 个值来初始化这些元素。这之所以可行，是因为容器类现在包含将 initializer_list<T>作为参数的构造函数。例如，vector<double>包含一个将 initializer_list<double>作为参数的构造函数，因此上述声明与下面的代码等价：

```
std::vector<double> payments({45.99, 39.23, 19.95, 89.01});
```

这里显式地将列表指定为构造函数参数。

通常，考虑到 C++11 新增的通用初始化语法，可使用表示法{}而不是()来调用类构造函数：

```
shared_ptr<double> pd {new double}; // ok to use {} instead of ()
```

但如果类也有接受 initializer_list 作为参数的构造函数，这将带来问题：

```
std::vector<int> vi{10};    // ??
```

这将调用哪个构造函数呢？

```
std::vector<int> vi(10);    // case A: 10 uninitialized elements
std::vector<int> vi({10});  // case B: 1 element set to 10
```

答案是，如果类有接受 initializer_list 作为参数的构造函数，则使用语法{}将调用该构造函数。因此在这个示例中，对应的是情形 B。

所有 initializer_list 元素的类型都必须相同，但编译器将进行必要的转换：

```
std::vector<double> payments {45.99, 39.23, 19, 89};
// same as std::vector<double> payments {45.99, 39.23, 19.0, 89.0};
```

在这里，由于 vector 的元素类型为 double，因此列表的类型为 initializer_list<double>，所以 19 和 89 被转换为 double。

但不能进行隐式的窄化转换：

```
std::vector<int> values = {10, 8, 5.5};  // narrowing, compile-time error
```

在这里，元素类型为 int，不能隐式地将 5.5 转换为 int。

除非类要用于处理长度不同的列表，否则让它提供接受 initializer_list 作为参数的构造函数没有意义。例如，对于存储固定数目值的类，您不想提供接受 initializer_list 作为参数的构造函数。在下面的声明中，类包含三个数据成员，因此没有提供 initializer_list 作为参数的构造函数：

```
class Position
{
private:
    int x;
    int y;
    int z;
public:
    Position(int xx = 0, int yy  = 0, int zz = 0)
            : x(xx), y(yy), z(zz) {}
    // no initializer_list constructor
    ...
};
```

这样，使用语法{}时将调用构造函数 Position(int, int, int)：

```
Position A = {20, -3};  // uses Position(20,-3,0)
```

16.7.3　使用 initializer_list

要在代码中使用 initializer_list 对象，必须包含头文件 initializer_list。这个模板类包含成员函数 begin( )和 end( )，您可使用这些函数来访问列表元素。它还包含成员函数 size( )，该函数返回元素数。程序清单 16.22 是一个简单的 initializer_list 使用示例，它要求编译器支持 C++11 新增的 initializer_list。

程序清单 16.22 ilist.cpp

```
// ilist.cpp -- use initializer_list (C++11 feature)
#include <iostream>
#include <initializer_list>

double sum(std::initializer_list<double> il);
double average(const std::initializer_list<double> & ril);

int main()
{
    using std::cout;

    cout << "List 1: sum = " << sum({2,3,4})
         <<", ave = " << average({2,3,4}) << '\n';
    std::initializer_list<double> dl = {1.1, 2.2, 3.3, 4.4, 5.5};
    cout << "List 2: sum = " << sum(dl)
         <<", ave = " << average(dl) << '\n';
    dl = {16.0, 25.0, 36.0, 40.0, 64.0};
    cout << "List 3: sum = " << sum(dl)
         <<", ave = " << average(dl) << '\n';
    return 0;
}

double sum(std::initializer_list<double> il)
{
    double tot = 0;
    for (auto p = il.begin(); p !=il.end(); p++)
        tot += *p;
    return tot;
}

double average(const std::initializer_list<double> & ril)
{
    double tot = 0;
    int n = ril.size();
    double ave = 0.0;
    if (n > 0)
    {
        for (auto p = ril.begin(); p !=ril.end(); p++)
            tot += *p;
        ave = tot / n;
    }
    return ave;
}
```

该程序的输出如下：

```
List 1: sum = 9, ave = 3
List 2: sum = 16.5, ave = 3.3
List 3: sum = 181, ave = 36.2
```

程序说明

可按值传递 initializer_list 对象，也可按引用传递，如 sum()和 average()所示。这种对象本身很小，通

常是两个指针（一个指向开头，一个指向末尾的下一个元素），也可能是一个指针和一个表示元素数的整数，因此采用的传递方式不会带来重大的性能影响。STL 按值传递它们。

函数参数可以是 initializer_list 字面量，如{2, 3, 4}，也可以是 initializer_list 变量，如 dl。

initializer_list 的迭代器类型为 const，因此您不能修改 initializer_list 中的值：

```
*dl.begin() = 2011.6;                // not allowed
```

但正如程序清单 16.22 演示的，可以将一个 initializer_list 赋给另一个 initializer_list：

```
dl = {16.0, 25.0, 36.0, 40.0, 64.0};  // allowed
```

然而，提供 initializer_list 类的初衷旨在让您能够将一系列值传递给构造函数或其他函数。

16.8　总结

C++提供了一组功能强大的库，这些库提供了很多常见编程问题的解决方案以及简化其他问题的工具。string 类为将字符串作为对象来处理提供了一种方便的方法。string 类提供了自动内存管理功能以及众多处理字符串的方法和函数。例如，这些方法和函数让您能够合并字符串、将一个字符串插入到另一个字符串中、反转字符串、在字符串中搜索字符或子字符串以及执行输入和输出操作。

诸如 auto_ptr 以及 C++11 新增的 shared_ptr 和 unique_ptr 等智能指针模板使得管理由 new 分配的内存更容易。如果使用这些智能指针（而不是常规指针）来保存 new 返回的地址，则不必在以后使用删除运算符。智能指针对象过期时，其析构函数将自动调用 delete 运算符。

STL 是一个容器类模板、迭代器类模板、函数对象模板和算法函数模板的集合，它们的设计是一致的，都是基于泛型编程原则的。算法通过使用模板，从而独立于所存储的对象的类型；通过使用迭代器接口，从而独立于容器的类型。迭代器是广义指针。

STL 使用术语“概念”来描述一组要求。例如，正向迭代器的概念包含这样的要求，即正向迭代器能够被解除引用，以便读写，同时能够被递增。概念真正的实现方式被称为概念的“模型”。例如，正向迭代器概念可以是常规指针或导航链表的对象。基于其他概念的概念叫作“改进”。例如，双向迭代器是正向迭代器概念的改进。

诸如 vector 和 set 等容器类是容器概念（如容器、序列和关联容器）的模型。STL 定义了多种容器类模板：vector、deque、list、set、multiset、map、multimap 和 bitset；还定义了适配器类模板 queue、priority_queue 和 stack；这些类让底层容器类能够提供适配器类模板名称所建议的特性接口。因此，stack 虽然在默认情况下是基于 vector 的，但仍只允许在栈顶进行插入和删除。C++11 新增了 forward_list、unordered_set、unordered_multiset、unordered_map 和 unordered_multimap。

有些算法被表示为容器类方法，但大量算法都被表示为通用的、非成员函数，这是通过将迭代器作为容器和算法之间的接口得以实现的。这种方法的一个优点是：只需一个诸如 for_each( )或 copy( )这样的函数，而不必为每种容器提供一个版本；另一个优点是：STL 算法可用于非 STL 容器，如常规数组、string 对象、array 对象以及您设计的秉承 STL 迭代器和容器规则的任何类。

容器和算法都是由其提供或需要的迭代器类型表征的。应当检查容器是否具备支持算法要求的迭代器概念。例如，for_each( )算法使用一个输入迭代器，所有的 STL 容器类类型都满足其最低要求；而 sort( )则要求随机访问迭代器，并非所有的容器类都支持这种迭代器。如果容器类不能满足特定算法的要求，则可能提供一个专用的方法。例如，list 类包含一个基于双向迭代器的 sort( )方法，因此它可以使用该方法，而不是通用函数。

STL 还提供了函数对象（函数符），函数对象是重载了( )运算符（即定义了 operator( )( )方法）的类。可以使用函数表示法来调用这种类的对象，同时可以携带额外的信息。自适应函数符有 typedef 语句，这种语句标识了函数符的参数类型和返回类型。这些信息可供其他组件（如函数适配器）使用。

通过表示常用的容器类型，并提供各种使用高效算法实现的常用操作（全部是通用的方式实现的），

STL 提供了一个非常好的可重用代码源。可以直接使用 STL 工具来解决编程问题，也可以把它们作为基本部件，来构建所需的解决方案。

模板类 complex 和 valarray 支持复数和数组的数值运算。

16.9 复习题

1．考虑下面的类声明：

```
class RQ1
{
private:
    char * st;        // points to C-style string
public:
    RQ1() { st = new char [1]; strcpy(st,""); }
    RQ1(const char * s)
    {st = new char [strlen(s) + 1]; strcpy(st, s); }
    RQ1(const RQ1 & rq)
    {st = new char [strlen(rq.st) + 1]; strcpy(st, rq.st); }
    ~RQ1() {delete [] st};
    RQ & operator=(const RQ & rq);
    // more stuff
};
```

将它转换为使用 string 对象的声明。哪些方法不再需要显式定义？

2．在易于使用方面，指出 string 对象至少两个优于 C-风格字符串的地方。

3．编写一个函数，用 string 对象作为参数，将 string 对象转换为全部大写。

4．从概念上或语法上说，下面哪个不是正确使用 auto_ptr 的方法（假设已经包含了所需的头文件）？

```
auto_ptr<int> pia(new int[20]);
auto_ptr<string> (new string);
int rigue = 7;
auto_ptr<int>pr(&rigue);
auto_ptr dbl (new double);
```

5．如果可以生成一个存储高尔夫球棍（而不是数字）的栈，为何它（从概念上说）是一个坏的高尔夫袋子？

6．为什么说对于逐洞记录高尔夫成绩来说，set 容器是糟糕的选择？

7．既然指针是一个迭代器，为什么 STL 设计人员没有简单地使用指针来代替迭代器呢？

8．为什么 STL 设计人员仅定义了迭代器基类，而使用继承来派生其他迭代器类型的类，并根据这些迭代器类来表示算法？

9．给出 vector 对象比常规数组方便的 3 个例子。

10．如果程序清单 16.9 是使用 list（而不是 vector）实现的，则该程序的哪些部分将是非法的？非法部分能够轻松修复吗？如果可以，如何修复呢？

11．假设有程序清单 16.15 所示的函数符 TooBig，下面的代码有何功能？赋给 bo 的是什么值？

```
bool bo = TooBig<int>(10)(15);
```

16.10 编程练习

1．回文指的是顺读和逆读都一样的字符串。例如，“tot”和“otto”都是简短的回文。编写一个程序，

让用户输入字符串，并将字符串引用传递给一个 bool 函数。如果字符串是回文，该函数将返回 true，否则返回 false。此时，不要担心诸如大小写、空格和标点符号这些复杂的问题。即这个简单的版本将拒绝“Otto”和“Madam，I'm Adam”。请查看附录 F 中的字符串方法列表，以简化这项任务。

2．与编程练习 1 中给出的问题相同，但要考虑诸如大小写、空格和标点符号这样的复杂问题。即“Madam，I'm Adam”将作为回文来测试。例如，测试函数可能会将字符串缩略为“madamimadam”，然后测试倒过来是否一样。不要忘了有用的 cctype 库，您可能从中找到几个有用的 STL 函数，尽管不一定非要使用它们。

3．修改程序清单 16.3，使之从文件中读取单词。一种方案是，使用 vector<string>对象而不是 string 数组。这样便可以使用 push_back( )将数据文件中的单词复制到 vector<string>对象中，并使用 size( )来确定单词列表的长度。由于程序应该每次从文件中读取一个单词，因此应使用运算符>>而不是 getline( )。文件中包含的单词应该用空格、制表符或换行符分隔。

4．编写一个具有老式风格接口的函数，其原型如下：

```
int reduce(long ar[], int n);
```

实参应是数组名和数组中的元素个数。该函数对数组进行排序，删除重复的值，返回缩减后数组中的元素数目。请使用 STL 函数编写该函数（如果决定使用通用的 unique( )函数，请注意它将返回结果区间的结尾）。使用一个小程序测试该函数。

5．问题与编程练习 4 相同，但要编写一个模板函数：

```
template <class T>
int reduce(T ar[], int n);
```

在一个使用 long 实例和 string 实例的小程序中测试该函数。

6．使用 STL queue 模板类而不是第 12 章的 Queue 类，重新编写程序清单 12.12 所示的示例。

7．彩票卡是一个常见的游戏。卡片上是带编号的圆点，其中一些圆点被随机选中。编写一个 lotto( )函数，它接受两个参数。第一个参数是彩票卡上圆点的个数，第二个参数是随机选择的圆点个数。该函数返回一个 vector<int>对象，其中包含（按排列后的顺序）随机选择的号码。例如，可以这样使用该函数：

```
vector<int> winners;
winners = Lotto(51,6);
```

这样将把一个矢量赋给 winner，该矢量包含 1～51 中随机选定的 6 个数字。注意，仅仅使用 rand( )无法完成这项任务，因它会生成重复的值。提示：让函数创建一个包含所有可能值的矢量，使用 random_shuffle( )，然后通过打乱后的矢量的第一个值来获取值。编写一个小程序来测试这个函数。

8．Mat 和 Pat 希望邀请他们的朋友来参加派对。他们要编写一个程序完成下面的任务。

- 让 Mat 输入他朋友的姓名列表。姓名存储在一个容器中，然后按排列后的顺序显示出来。
- 让 Pat 输入她朋友的姓名列表。姓名存储在另一个容器中，然后按排列后的顺序显示出来。
- 创建第三个容器，将两个列表合并，删除重复的部分，并显示这个容器的内容。

9．相对于数组，在链表中添加和删除元素更容易，但排序速度更慢。这就引出了一种可能性：相对于使用链表算法进行排序，将链表复制到数组中，对数组进行排序，再将排序后的结果复制到链表中的速度可能更快；但这也可能占用更多的内存。请使用如下方法检验上述假设。

a．创建大型 vector<int>对象 vi0，并使用 rand( )给它提供初始值。

b．创建 vector<int>对象 vi 和 list<int>对象 li，它们的长度都和初始值与 vi0 相同。

c．计算使用 STL 算法 sort( )对 vi 进行排序所需的时间，再计算使用 list 的方法 sort( )对 li 进行排序所需的时间。

d．将 li 重置为排序的 vi0 的内容，并计算执行如下操作所需的时间：将 li 的内容复制到 vi 中，对 vi 进行排序，并将结果复制到 li 中。

要计算这些操作所需的时间，可使用 ctime 库中的 clock( )。正如程序清单 5.14 演示的，可使用下面的语句来获取开始时间：

```
clock_t start = clock();
```

再在操作结束后使用下面的语句获取经过了多长时间：

```
clock_t end = clock();
cout << (double)(end - start)/CLOCKS_PER_SEC;
```

这种测试并非绝对可靠，因为结果取决于很多因素，如可用内存量、是否支持多处理以及数组（列表）的长度（随着要排序的元素数增加，数组相对于列表的效率将更明显）。另外，如果编译器提供了默认生成方式和发布生成方式，请使用发布生成方式。鉴于当今计算机的速度非常快，要获得有意义的结果，可能需要使用尽可能大的数组。例如，可尝试包含 100000、1000000 和 10000000 个元素。

10．请按如下方式修改程序清单 16.9（vect3.cpp）。

a．在结构 Review 中添加成员 price。

b．不使用 vector<Review>来存储输入，而使用 vector<shared_ptr<Review>>。别忘了，必须使用 new 返回的指针来初始化 shared_ptr。

c．在输入阶段结束后，使用一个循环让用户选择如下方式之一显示书籍：按原始顺序显示、按字母表顺序显示、按评级升序显示、按评级降序显示、按价格升序显示、按价格降序显示、退出。

下面是一种可能的解决方案：获取输入后，再创建一个 shared_ptr 矢量，并用原始数组初始化它。定义一个对指向结构的指针进行比较的 operator<( )函数，并使用它对第二个矢量进行排序，让其中的 shared_ptr 按其指向的对象中的书名排序。重复上述过程，创建按 rating 和 price 排序的 shared_ptr 矢量。请注意，通过使用 rbegin()和 rend()，可避免创建按相反的顺序排列的 shared_ptr 矢量。

第 17 章　输入、输出和文件

本章内容包括：

- C++角度的输入和输出。
- iostream 类系列。
- 重定向。
- ostream 类方法。
- 格式化输出。
- istream 类方法。
- 流状态。
- 文件 I/O。
- 使用 ifstream 类从文件输入。
- 使用 ofstream 类输出到文件。
- 使用 fstream 类进行文件输入和输出。
- 命令行处理。
- 二进制文件。
- 随机文件访问。
- 内核格式化。

对 C++输入和输出（简称 I/O）的讨论提出了一个问题。一方面，几乎每个程序都要使用输入和输出，因此了解如何使用它们是每个学习计算机语言的人面临的首要任务；另一方面，C++使用了很多较为高级的语言特性来实现输入和输出，其中包括类、派生类、函数重载、虚函数、模板和多重继承。因此，要真正理解 C++ I/O，必须了解 C++的很多内容。为了帮助您起步，本书的开始几章介绍了使用 istream 类对象 cin 和 ostream 类对象 cout 进行输入和输出的基本方法，同时使用了 ifstream 和 ofstream 对象进行文件输入和输出。本章将更详细地介绍 C++的输入和输出类，看看它们是如何设计的，学习如何控制输出格式（如果您跳过很多章，直接学习高级格式，可浏览一下讨论该主题的一些小节，注意其中的技术，而忽略解释）。

用于文件输入和输出的 C++工具都是基于 cin 和 cout 所基于的基本类定义，因此本章以对控制台 I/O（键盘和屏幕）的讨论为跳板，来研究文件 I/O。

ANSI/ISO C++标准委员会的工作是让 C++ I/O 与现有的 C I/O 更加兼容，这给传统的 C++做法带来了一些变化。

17.1　C++输入和输出概述

多数计算机语言的输入和输出是以语言本身为基础实现的。例如，从诸如 BASIC 和 Pascal 等语言的关键字列表中可知，PRINT 语句、Writeln 语句以及其他类似的语句都是语言词汇表的组成部分，但 C 和 C++都

没有将输入和输出建立在语言中。这两种语言的关键字包括 for 和 if，但不包括与 I/O 有关的内容。C 语言最初把 I/O 留给了编译器实现人员。这样做的一个原因是为了让实现人员能够自由的设计 I/O 函数，使之最适合于目标计算机的硬件要求。实际上，多数实现人员都把 I/O 建立在最初为 UNIX 环境开发的库函数的基础之上。ANSI C 正式承认这个 I/O 软件包时，将其称为标准输入/输出包，并将其作为标准 C 库不可或缺的组成部分。C++也认可这个软件包，因此如果熟悉 stdio.h 文件中声明的 C 函数系列，则可以在 C++程序中使用它们（较新的实现使用头文件 cstdio 来支持这些函数）。

然而，C++依赖于 C++的 I/O 解决方案，而不是 C 语言的 I/O 解决方案，前者是在头文件 iostream（以前为 iostream.h）和 fstream（以前为 fstream.h）中定义一组类。这个类库不是正式语言定义的组成部分（cin 和 istream 不是关键字）；毕竟计算机语言定义了如何工作（例如如何创建类）的规则，但没有定义应按照这些规则创建哪些东西。然而，正如 C 实现自带了一个标准函数库一样，C++也自带了一个标准类库。首先，标准类库是一个非正式的标准，只是由头文件 iostream 和 fstream 中定义的类组成。ANSI/ISO C++委员会决定把这个类正式作为一个标准类库，并添加其他一些标准类，如第 16 章讨论的那些类。本章将讨论标准 C++ I/O。但首先看一看 C++ I/O 的概念框架。

17.1.1 流和缓冲区

C++程序把输入和输出看作字节流。输入时，程序从输入流中抽取字节；输出时，程序将字节插入到输出流中。对于面向文本的程序，每个字节代表一个字符，更通俗地说，字节可以构成字符或数值数据的二进制表示。输入流中的字节可能来自键盘，也可能来自存储设备（如硬盘）或其他程序。同样，输出流中的字节可以流向屏幕、打印机、存储设备或其他程序。流充当了程序和流源或流目标之间的桥梁。这使得 C++程序可以以相同的方式对待来自键盘的输入和来自文件的输入。C++程序只是检查字节流，而不需要知道字节来自何方。同理，通过使用流，C++程序处理输出的方式将独立于其去向。因此管理输入包含两步：

- 将流与输入去向的程序关联起来。
- 将流与文件连接起来。

换句话说，输入流需要两个连接，每端各一个。文件端部连接提供了流的来源，程序端连接将流的流出部分转储到程序中（文件端连接可以是文件，也可以是设备，如键盘）。同样，对输出的管理包括将输出流连接到程序以及将输出目标与流关联起来。这就像将字节（而不是水）引入到水管中（参见图 17.1）。

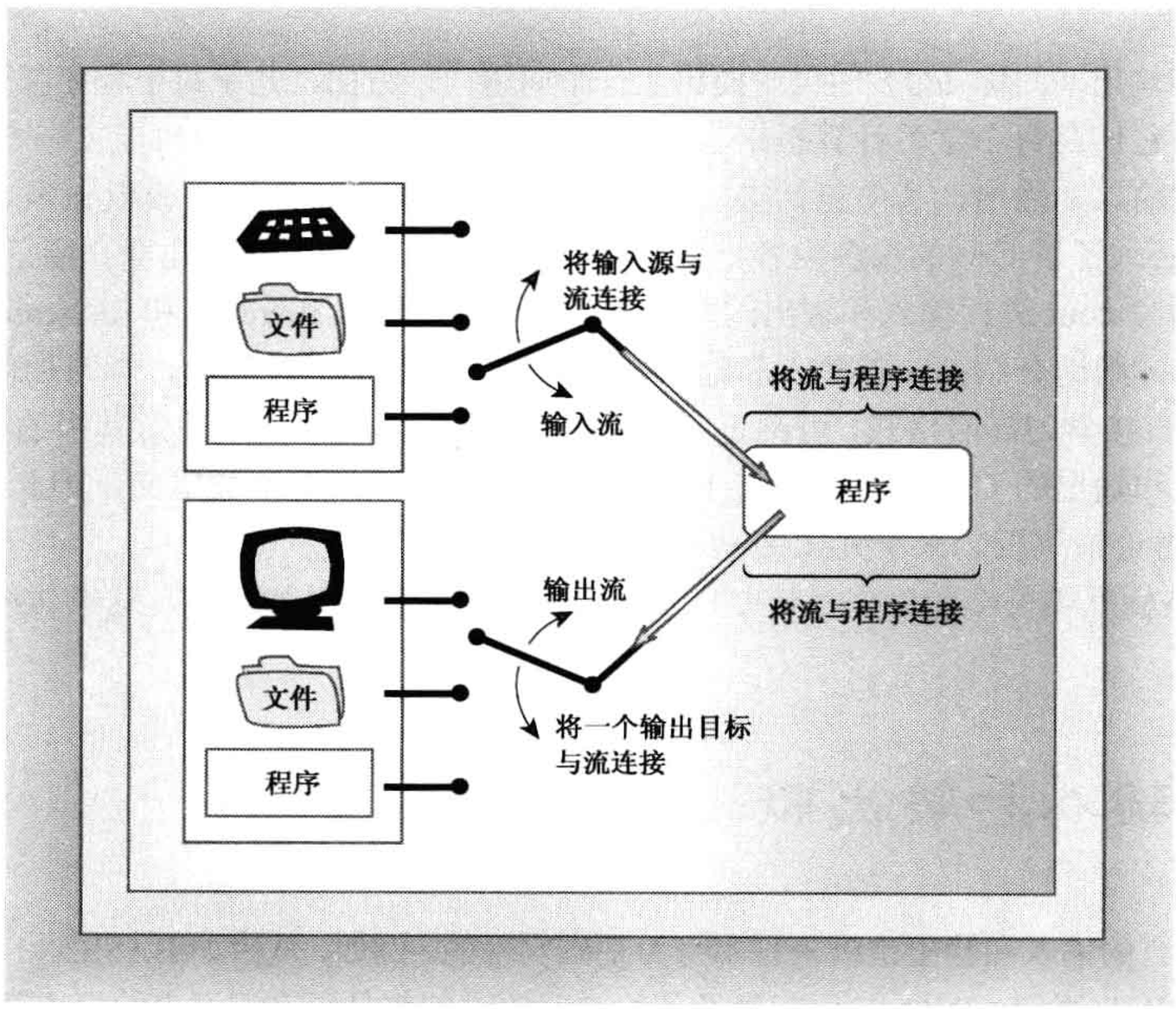

图 17.1 C++输入和输出

通常，通过使用缓冲区可以更高效地处理输入和输出。缓冲区是用作中介的内存块，它是将信息从设备传输到程序或从程序传输给设备的临时存储工具。通常，像磁盘驱动器这样的设备以 512 字节（或更多）的块为单位来传输信息，而程序通常每次只能处理一个字节的信息。缓冲区帮助匹配这两种不同的信息传输速率。例如，假设程序要计算记录在硬盘文件中的金额。程序可以从文件中读取一个字符，处理它，再从文件中读取下一个字符，再处理，依此类推。从磁盘文件中每次读取一个字符需要大量的硬件活动，速度非常慢。缓冲方法则从磁盘上读取大量信息，将这些信息存储在缓冲区中，然后每次从缓冲区里读取一个字节。因为从内存中读取单个字节的速度比从硬盘上读取快很多，所以这种方法更快，也更方便。当然，到达缓冲区尾部后，程序将从磁盘上读取另一块数据。这种原理与水库在暴风雨中收集几兆加仑流量的水，然后以比较文明的速度给您家里供水是一样的（见图 17.2）。输出时，程序首先填满缓冲区，然后把整块数据传输给硬盘，并清空缓冲区，以备下一批输出使用。这被称为刷新缓冲区（flushing the buffer）。

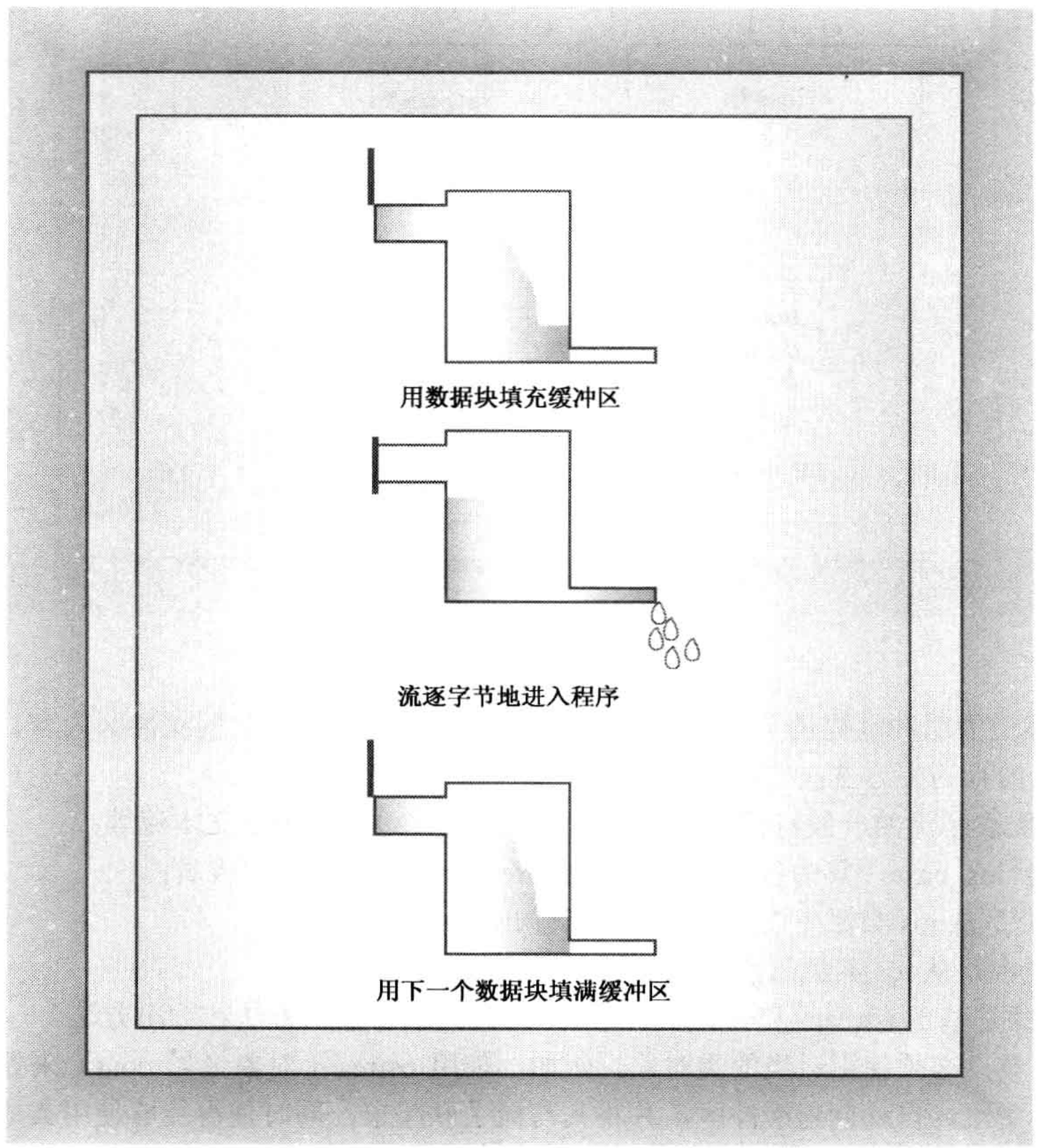

图 17.2　有缓冲区的流

键盘输入每次提供一个字符，因此在这种情况下，程序无需缓冲区来帮助匹配不同的数据传输速率。然而，对键盘输入进行缓冲可以让用户在将输入传输给程序之前返回并更正。C++程序通常在用户按下回车键时刷新输入缓冲区。这是为什么本书的例子没有一开始就处理输入，而是等到用户按下回车键后再处理的原因。对于屏幕输出，C++程序通常在用户发送换行符时刷新输出缓冲区。程序也可能会在其他情况下刷新输入，例如输入即将到来时，这取决于实现。也就是说，当程序到达输入语句时，它将刷新输出缓冲区中当前所有的输出。与 ANSI C 一致的 C++实现是这样工作的。

17.1.2　流、缓冲区和 iostream 文件

管理流和缓冲区的工作有点复杂，但 iostream（以前为 iostream.h）文件中包含一些专门设计用来实现、

管理流和缓冲区的类。C++98 版本 C++ I/O 定义了一些类模板，以支持 char 和 wchar_t 数据；C++11 添加了 char16_t 和 char32_t 具体化。通过使用 typedef 工具，C++使得这些模板 char 具体化能够模仿传统的非模板 I/O 实现。下面是其中的一些类（见图 17.3）：

图 17.3 一些 I/O 类

- streambuf 类为缓冲区提供了内存，并提供了用于填充缓冲区、访问缓冲区内容、刷新缓冲区和管理缓冲区内存的类方法；
- ios_base 类表示流的一般特征，如是否可读取、是二进制流还是文本流等；
- ios 类基于 ios_base，其中包括了一个指向 streambuf 对象的指针成员；
- ostream 类是从 ios 类派生而来的，提供了输出方法；
- istream 类也是从 ios 类派生而来的，提供了输入方法；
- iostream 类是基于 istream 和 ostream 类的，因此继承了输入方法和输出方法。

要使用这些工具，必须使用适当的类对象。例如，使用 ostream 对象（如 cout）来处理输出。创建这样的对象将打开一个流，自动创建缓冲区，并将其与流关联起来，同时使得能够使用类成员函数。

重定义 I/O

ISO/ANSI 标准 C++98 对 I/O 作了两方面的修订。首先是从 ostream.h 到 ostream 的变化，用 ostream 将类放到 std 名称空间中。其次，I/O 类被重新编写。为成为国际语言，C++必须能够处理需要 16 位的国际字符集或更宽的字符类型。因此，该语言在传统的 8 位 char（“窄”）类型的基础上添加了 wchar_t（“宽”）字符类型；而 C++11 添加了类型 char16_t 和 char32_t。每种类型都需要有自己的 I/O 工具。标准委员会并没有开发两套（现在为 4 套）独立的类，而是开发了 1 套 I/O 类模板，其中包括 basic_istream<charT, traits<charT>>和 basic_ostream<charT, traits<charT>>。traits<charT>模板是一个模板类，为字符类型定义了具体特性，如如何比较字符是否相等以及字符的 EOF 值等。该 C++11 标准提供了 I/O 的 char 和 wchar_t 具体化。例如，istream 和 ostream 都是 char 具体化的 typedef。同样，wistream 和 wostream 都是 wchar_t 具体化。例如，wcout 对象用于输出宽字符流。头文件 ostream 中包含了这些定义。

ios 基类中的一些独立于类型的信息被移动到新的 ios_base 类中，这包括各种格式化常量，例如 ios::fixed（现在为 ios_base::fixed）。另外，ios_base 还包含了一些老式 ios 中没有的选项。

C++的 iostream 类库管理了很多细节。例如，在程序中包含 iostream 文件将自动创建 8 个流对象（4 个用于窄字符流，4 个用于宽字符流）。

- cin 对象对应于标准输入流。在默认情况下，这个流被关联到标准输入设备（通常为键盘）。wcin 对象与此类似，但处理的是 wchar_t 类型。
- cout 对象与标准输出流相对应。在默认情况下，这个流被关联到标准输出设备（通常为显示器）。wcout 对象与此类似，但处理的是 wchar_t 类型。
- cerr 对象与标准错误流相对应，可用于显示错误消息。在默认情况下，这个流被关联到标准输出设备（通常为显示器）。这个流没有被缓冲，这意味着信息将被直接发送给屏幕，而不会等到缓冲区填满或新的换行符。wcerr 对象与此类似，但处理的是 wchar_t 类型。
- clog 对象也对应着标准错误流。在默认情况下，这个流被关联到标准输出设备（通常为显示器）。这个流被缓冲。wclog 对象与此类似，但处理的是 wchar_t 类型。
- 对象代表流——这意味着什么呢？当 iostream 文件为程序声明一个 cout 对象时，该对象将包含存储了与输出有关的信息的数据成员，如显示数据时使用的字段宽度、小数位数、显示整数时采用的计数方法以及描述用来处理输出流的缓冲区的 streambuf 对象的地址。下面的语句通过指向的 streambuf 对象将字符串“Bjarna free”中的字符放到 cout 管理的缓冲区中：

```
cout << "Bjarne free";
```

ostream 类定义了上述语句中使用的 operator<<( )函数，ostream 类还支持 cout 数据成员以及其他大量的类方法（如本章稍后将讨论的那些方法）。另外，C++注意到，来自缓冲区的输出被导引到标准输出（通常是显示器，由操作系统提供）。总之，流的一端与程序相连，另一端与标准输出相连，cout 对象凭借 streambuf 对象的帮助，管理着流中的字节流。

17.1.3　重定向

标准输入和输出流通常连接着键盘和屏幕。但很多操作系统（包括 UNIX、Linux 和 Windows）都支持重定向，这个工具使得能够改变标准输入和标准输出。例如，假设有一个名为 counter.exe 的、可执行的 Windows 命令提示符 C++程序，它能够计算输入中的字符数，并报告结果（在大多数 Windows 系统中，可以选择“开始”>“程序”，再单击“命令提示符”来打开命令提示符窗口）。该程序的运行情况如下：

```
C>counter
Hello
and goodbye!
Control-Z            << simulated end-of-file
Input contained 19 characters.
C>
```

其中的输入来自键盘，输出的被显示到屏幕上。

通过输入重定向（<）和输出重定向（>），可以使用上述程序计算文件 oklahoma 中的字符数，并将结果放到 cow_cnt 文件中：

```
cow_cnt file:
C>counter <oklahoma >cow_cnt
C>
```

命令行中的<oklahoma 将标准输入与 oklahoma 文件关联起来，使 cin 从该文件（而不是键盘）读取输入。换句话说，操作系统改变了输入流的流入端连接，而流出端仍然与程序相连。命令行中的>cow_cnt 将标准输出与 cow_cnt 文件关联起来，导致 cout 将输出发送给文件（而不是屏幕）。也就是说，操作系统改变了输出流的流出端连接，而流入端仍与程序相连。DOS、Windows 命令提示符模式、Linux 和 UNIX 能自动识别这种重定向语法（除早期的 DOS 外，其他操作系统都允许在重定向运算符与文件名之间加上可选

的空格）。

cout 代表的标准输出流是程序输出的常用通道。标准错误流（由 cerr 和 clog 代表）用于程序的错误消息。默认情况下，这 3 个对象都被发送给显示器。但对标准输出重定向并不会影响 cerr 或 clog，因此，如果使用其中一个对象来打印错误消息，程序将在屏幕上显示错误消息，即使常规的 cout 输出被重定向到其他地方。例如，请看下面的代码片段：

```
if (success)
    std::cout << "Here come the goodies!\n";
else
{
    std::cerr << "Something horrible has happened.\n";
    exit(1);
}
```

如果重定向没有起作用，则选定的消息都将被显示在屏幕上。然而，如果程序输出被重定向到一个文件，则第一条消息（如果被选定）将被发送到文件中，而第二条消息（如果被选定）将被发送到屏幕。顺便说一句，有些操作系统也允许对标准错误进行重定向。例如，在 UNIX 和 Linux 中，运算符 2>重定向标准错误。

17.2 使用 cout 进行输出

正如前面指出的，C++将输出看作字节流（根据实现和平台的不同，可能是 8 位、16 位或 32 位的字节，但都是字节），但在程序中，很多数据被组织成比字节更大的单位。例如，int 类型由 16 位或 32 位的二进制值表示；double 值由 64 位的二进制数据表示。但在将字节流发送给屏幕时，希望每个字节表示一个字符值。也就是说，要在屏幕上显示数字–2.34，需要将 5 个字符（–、2、.、3 和 4），而不是这个值的 64 位内部浮点表示发送到屏幕上。因此，ostream 类最重要的任务之一是将数值类型（如 int 或 float）转换为以文本形式表示的字符流。也就是说，ostream 类将数据内部表示（二进制位模式）转换为由字符字节组成的输出流（以后会有仿生移植物，使得能够直接翻译二进制数据。我们把这种开发作为一个练习，留给您）。为执行这些转换任务，ostream 类提供了多个类方法。现在就来看看它们，总结本书使用的方法，并介绍能够更精密地控制输出外观的其他方法。

17.2.1 重载的<<运算符

本书常结合使用 cout 和<<运算符（插入（insertion）运算符）：

```
int clients = 22;
cout << clients;
```

在 C++中，与 C 一样，<<运算符的默认含义是按位左移运算符（参见附录 E）。表达式 x<<3 的意思，将 x 的二进制表示中所有的位向左移动 3 位。显然，这与输出的关系不大。但 ostream 类重新定义了<<运算符，方法是将其重载为输出。在这种情况下，<<叫作插入运算符，而不是左移运算符（左移运算符由于其外观（像向左流动的信息流）而获得这种新角色）。插入运算符被重载，使之能够识别 C++中所有的基本类型：

- unsigned char；
- signed char；
- char；
- short；
- unsigned short；
- int；

- unsiged int；
- long；
- unsigned long；
- long long（C++11）；
- unsigned long long（C++11）；
- float；
- double；
- long double。

对于上述每种数据类型，ostream 类都提供了 operator<<( )函数的定义（第 11 章讨论过，名称中包含运算符的函数用于重载该运算符）。因此，如果使用下面这样一条语句，而 value 是前面列出的类型之一，则 C++程序将其对应于有相应的特征标的运算符函数：

```
cout << value;
```

例如，表达式 cout<<88 对应于下面的方法原型：

```
ostream & operator<<(int);
```

该原型表明，operator<<( )函数接受一个 int 参数，这与上述语句中的 88 匹配。该原型还表明，函数返回一个指向 ostream 对象的引用，这使得可以将输出连接起来，如下所示：

```
cout << "I'm feeling sedimental over " << boundary << "\n";
```

如果您是 C 语言程序员，深受%类型说明符过多、说明符类型与值不匹配时将发生问题等痛苦，则使用 cout 非常简单（当然，由于有 cin，C++输入也非常简单）。

1. 输出和指针

ostream 类还为下面的指针类型定义了插入运算符函数：

- const signed char *；
- const unsigned char *；
- const char *；
- void *。

不要忘了，C++用指向字符串存储位置的指针来表示字符串。指针的形式可以是 char 数组名、显式的 char 指针或用引号括起的字符串。因此，下面所有的 cout 语句都显示字符串：

```
char name[20] = "Dudly Diddlemore";
char * pn = "Violet D'Amore";
cout << "Hello!";
cout << name;
cout << pn;
```

方法使用字符串中的终止空字符来确定何时停止显示字符。

对于其他类型的指针，C++将其对应于 void *，并打印地址的数值表示。如果要获得字符串的地址，则必须将其强制转换为其他类型，如下面的代码片段所示：

```
int eggs = 12;
char * amount = "dozen";
cout << &eggs;              // prints address of eggs variable
cout << amount;             // prints the string "dozen"
cout << (void *) amount;    // prints the address of the "dozen" string
```

2. 拼接输出

插入运算符的所有化身的返回类型都是 ostream &。也就是说，原型的格式如下：

```
ostream & operator<<(type);
```

（其中，type 是要显示的数据的类型）返回类型 ostream &意味着使用该运算符将返回一个指向 ostream

对象的引用。哪个对象呢？函数定义指出，引用将指向用于调用该运算符的对象。换句话说，运算符函数的返回值为调用运算符的对象。例如，cout << “potluck”返回的是 cout 对象。这种特性使得能够通过插入来连接输出。例如，请看下面的语句：

```
cout << "We have " << count << " unhatched chickens.\n";
```

表达式 cout << “We have”将显示字符串，并返回 cout 对象。至此，上述语句将变为：

```
cout << count << " unhatched chickens.\n";
```

表达式 cout<<count 将显示 count 变量的值，并返回 cout。然后 cout 将处理语句中的最后一个参数（参见图 17.4）。这种设计技术确实是一项很好的特性，这也是前几章中重载<<运算符的示例模仿了这种技术的原因所在。

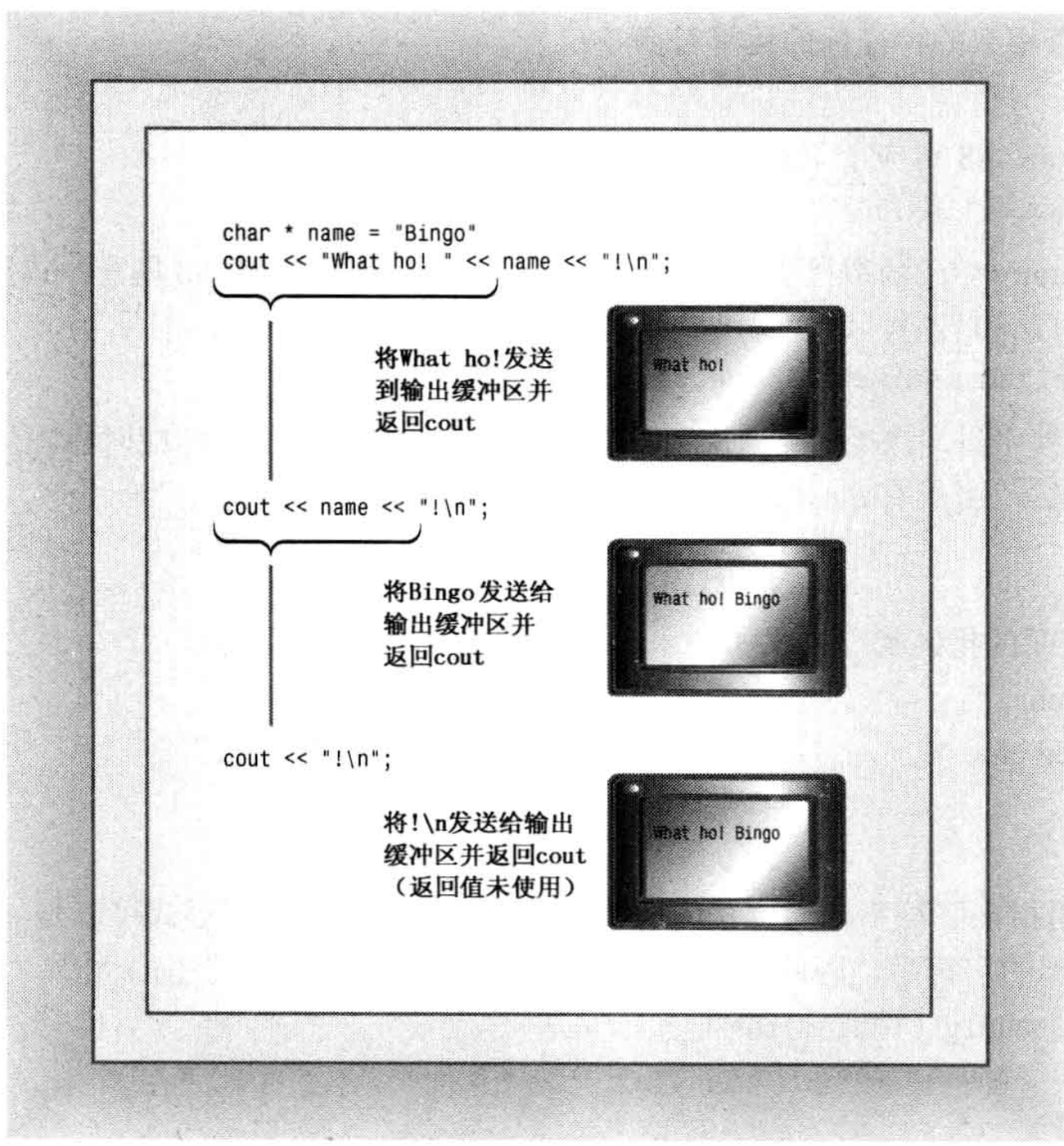

图 17.4 拼接输出

17.2.2 其他 ostream 方法

除了各种 operator<<( )函数外，ostream 类还提供了 put( )方法和 write( )方法，前者用于显示字符，后者用于显示字符串。

最初，put( )方法的原型如下：

```
ostream & put(char);
```

当前标准与此相同，但被模板化，以适用于 wchar_t。可以用类方法表示法来调用它：

```
cout.put('W');      // display the W character
```

其中，cout 是调用方法的对象，put( )是类成员函数。和<<运算符函数一样，该函数也返回一个指向调用对象的引用，因此可以用它将拼接输出：

```
cout.put('I').put('t'); // displaying It with two put() calls
```

函数调用 cout.put('I')返回 cout，cout 然后被用作 put('t')调用的调用对象。

在原型合适的情况下，可以将数值型参数（如 int）用于 put( )，让函数原型自动将参数转换为正确 char 值。例如，可以这样做：

```
cout.put(65);          // display the A character
cout.put(66.3);        // display the B character
```

第一条语句将 int 值 65 转换为一个 char 值，然后显示 ASCII 码为 65 的字符。同样，第二条语句将 double 值 66.3 转换为 char 值 66，并显示对应的字符。

这种行为在 C++ 2.0 之前可派上用场。在这些版本中，C++语言用 int 值表示字符常量。因此，下面的语句将'W'解释为一个 int 值，因此将其作为整数 87（即该字符的 ASCII 值）显示出来：

```
cout << 'W';
```

然而，下面这条语句能够正常工作：

```
cout.put('W');
```

因为当前的 C++将 char 常量表示为 char 类型，因此现在可以使用上述任何一种方法。

一些老式编译器错误地为 char、unsigned char 和 signed char 3 种参数类型重载了 put( )。这使得将 int 参数用于 put( )时具有二义性，因为 int 可被转换为这 3 种类型中的任何一种。

write( )方法显示整个字符串，其模板原型如下：

```
basic_ostream<charT,traits>& write(const char_type* s, streamsize n);
```

write( )的第一个参数提供了要显示的字符串的地址，第二个参数指出要显示多少个字符。使用 cout 调用 write( )时，将调用 char 具体化，因此返回类型为 ostream &。程序清单 17.1 演示了 write( )方法是如何工作的。

程序清单 17.1　write.cpp

```
// write.cpp -- using cout.write()
#include <iostream>
#include <cstring>  // or else string.h

int main()
{
    using std::cout;
    using std::endl;
    const char * state1 = "Florida";
    const char * state2 = "Kansas";
    const char * state3 = "Euphoria";
    int len = std::strlen(state2);
    cout << "Increasing loop index:\n";
    int i;
    for (i = 1; i <= len; i++)
    {
        cout.write(state2,i);
        cout << endl;
    }
// concatenate output
    cout << "Decreasing loop index:\n";
    for (i = len; i > 0; i--)
        cout.write(state2,i) << endl;

// exceed string length
    cout << "Exceeding string length:\n";
    cout.write(state2, len + 5) << endl;

    return 0;
}
```

有些编译器可能指出该程序定义了数组 state1 和 state3 但没有使用它们。这不是什么问题，因为这两个数组只是用于提供数组 state2 前面和后面的数据，以便您知道程序错误地存取 state2 时发生的情况。下面是程序清单 17.1 中程序的输出：

```
Increasing loop index:
K
Ka
Kan
Kans
Kansa
Kansas
Decreasing loop index:
Kansas
Kansa
Kans
Kan
Ka
K
Exceeding string length:
Kansas Euph
```

注意，cout.write( )调用返回 cout 对象。这是因为 write( )方法返回一个指向调用它的对象的引用，这里调用它的对象是 cout。

这使得可以将输出拼接起来，因为 cout.write( )将被其返回值 cout 替换：

```
cout.write(state2,i) << endl;
```

还需要注意的是，write( )方法并不会在遇到空字符时自动停止打印字符，而只是打印指定数目的字符，即使超出了字符串的边界！在这个例子中，在字符串“kansas”的前后声明了另外两个字符串，以便相邻的内存包含数据。编译器在内存中存储数据的顺序以及调整内存的方式各不相同。例如，“Kansas”占用 6 个字节，而该编译器使用 4 个字节的倍数调整字符串，因此“Kansas”被填充成占用 8 个字节。由于编译器之间的差别，因此输出的最后一行可能不同。

write( )方法也可用于数值数据，您可以将数字的地址强制转换为 char *，然后传递给它：

```
long val = 560031841;
cout.write( (char *) &val, sizeof (long));
```

这不会将数字转换为相应的字符，而是传输内存中存储的位表示。例如，4 字节的 long 值（如 560031841）将作为 4 个独立的字节被传输。输出设备（如显示器）将把每个字节作为 ASCII 码进行解释。因此在屏幕上，560031841 将被显示为 4 个字符的组合，这很可能是乱码（也可能不是，请试试看）。然而，write( )确实为将数值数据存储在文件中提供了一种简洁、准确的方式，这将在本章后面进行介绍。

17.2.3 刷新输出缓冲区

如果程序使用 cout 将字节发送给标准输出，情况将如何？由于 ostream 类对 cout 对象处理的输出进行缓冲，所以输出不会立即发送到目标地址，而是被存储在缓冲区中，直到缓冲区填满。然后，程序将刷新（flush）缓冲区，把内容发送出去，并清空缓冲区，以存储新的数据。通常，缓冲区为 512 字节或其整数倍。当标准输出连接的是硬盘上的文件时，缓冲可以节省大量的时间。毕竟，不希望程序为发送 512 个字节，而存取磁盘 512 次。将 512 个字节收集到缓冲区中，然后一次性将它们写入硬盘的效率要高得多。

然而，对于屏幕输出来说，首先填充缓冲区的重要性要低得多。如果必须重述消息“Press any key to continue”以便使用 512 个字节来填充缓冲区，实在是太不方便了。所幸的是，在屏幕输出时，程序不必等到缓冲区被填满。例如，将换行符发送到缓冲区后，将刷新缓冲区。另外，正如前面指出的，多数 C++实现都会在输入即将发生时刷新缓冲区。也就是说，假设有下面的代码：

```
cout << "Enter a number: ";
float num;
cin >> num;
```

程序期待输入这一事实，将导致它立刻显示 cout 消息（即刷新“Enter a number:”消息），即使输出字符串中没有换行符。如果没有这种特性，程序将等待输入，而无法通过 cout 消息来提示用户。

如果实现不能在所希望时刷新输出，可以使用两个控制符中的一个来强行进行刷新。控制符 flush 刷新缓冲区，而控制符 endl 刷新缓冲区，并插入一个换行符。这两个控制符的使用方式与变量名相同：

```
cout << "Hello, good-looking! " << flush;
cout << "Wait just a moment, please." << endl;
```

事实上，控制符也是函数。例如，可以直接调用 flush( )来刷新 cout 缓冲区：

```
flush(cout);
```

然而，ostream 类对<<插入运算符进行了重载，使得下述表达式将被替换为函数调用 flush(cout)：

```
cout << flush
```

因此，可以用更为方便的插入表示法来成功地进行刷新。

17.2.4　用 cout 进行格式化

ostream 插入运算符将值转换为文本格式。在默认情况下，格式化值的方式如下。

- 对于 char 值，如果它代表的是可打印字符，则将被作为一个字符显示在宽度为一个字符的字段中。
- 对于数值整型，将以十进制方式显示在一个刚好容纳该数字及负号（如果有的话）的字段中。
- 字符串被显示在宽度等于该字符串长度的字段中。

浮点数的默认行为有变化。下面详细说明了老式实现和新实现之间的区别。

- 新式：浮点类型被显示为 6 位，末尾的 0 不显示（注意，显示的数字位数与数字被存储时精度没有任何关系）。数字以定点表示法显示还是以科学计数法表示（参见第 3 章），取决于它的值。具体来说，当指数大于等于 6 或小于等于−5 时，将使用科学计数法表示。另外，字段宽度恰好容纳数字和负号（如果有的话）。默认的行为对应于带%g 说明符的标准 C 库函数 fprintf( )。
- 老式：浮点类型显示为带 6 位小数，末尾的 0 不显示（注意，显示的数字位数与数字被存储时的精度没有任何关系）。数字以定点表示法显示还是以科学计数法表示（参见第 3 章），取决于它的值。另外，字段宽度恰好容纳数字和负号（如果有的话）。

因为每个值的显示宽度都等于它的长度，因此必须显式地在值之间提供空格；否则，相邻的值将不会被分开。

程序清单 17.2 演示默认的输出情况，它在每个值后面都显示一个冒号（:），以便可以知道每种情况下的字段宽度。该程序使用表达式 1.0/9.0 来生成一个无穷小数，以便能够知道打印了多少位。

注意：并非所有的编译器都能生成符合当前 C++标准格式的输出。另外，当前标准允许区域性变化。例如，欧洲实现可能遵循欧洲人的风格：使用逗号而不是句点来表示小数点。也就是说，2.54 将被写成 2，54。区域库（头文件 locale）提供了用特定的风格影响（imbuing）输入或输出流的机制，所以同一个编译器能够提供多个区域选项。本章使用美国格式。

程序清单 17.2　defaults.cpp

```
// defaults.cpp -- cout default formats
#include <iostream>

int main()
{
    using std::cout;
    cout << "12345678901234567890\n";
```

```
    char ch = 'K';
    int t = 273;
    cout << ch << ":\n";
    cout << t << ":\n";
    cout << -t <<":\n";

    double f1 = 1.200;
    cout << f1 << ":\n";
    cout << (f1 + 1.0 / 9.0) << ":\n";

    double f2 = 1.67E2;
    cout << f2 << ":\n";
    f2 += 1.0 / 9.0;
    cout << f2 << ":\n";
    cout << (f2 * 1.0e4) << ":\n";


    double f3 = 2.3e-4;
    cout << f3 << ":\n";
    cout << f3 / 10 << ":\n";

    return 0;
}
```

程序清单 17.2 中程序的输出如下：

```
12345678901234567890
K:
273:
-273:
1.2:
1.31111:
167:
167.111:
1.67111e+006:
0.00023:
2.3e-005:
```

每个值都填充自己的字段。注意，1.200 末尾的 0 没有显示出来，但末尾不带 0 的浮点值后面将有 6 个空格。另外，该实现将指数显示为 3 位，而其他实现可能为两位。

1. 修改显示时使用的计数系统

ostream 类是从 ios 类派生而来的，而后者是从 ios_base 类派生而来的。ios_base 类存储了描述格式状态的信息。例如，一个类成员中某些位决定了使用的计数系统，而另一个成员则决定了字段宽度。通过使用控制符（manipulator），可以控制显示整数时使用的计数系统。通过使用 ios_base 的成员函数，可以控制字段宽度和小数位数。由于 ios_base 类是 ostream 的间接基类，因此可以将其方法用于 ostream 对象（或子代），如 cout。

注意： *ios_base 类中的成员和方法以前位于 ios 类中。现在，ios_base 是 ios 的基类。在新系统中，ios 是包含 char 和 wchar_t 具体化的模板，而 ios_base 包含了非模板特性。*

来看如何设置显示整数时使用的计数系统。要控制整数以十进制、十六进制还是八进制显示，可以使用 dec、hex 和 oct 控制符。例如，下面的函数调用将 cout 对象的计数系统格式状态设置为十六进制：

```
hex(cout);
```

完成上述设置后，程序将以十六进制形式打印整数值，直到将格式状态设置为其他选项为止。注意，控制符不是成员函数，因此不必通过对象来调用。

虽然控制符实际上是函数，但它们通常的使用方式为：

```
cout << hex;
```

ostream 类重载了<<运算符，这使得上述用法与函数调用 hex（cout）等价。控制符位于名称空间 std 中。程序清单 17.3 演示了这些控制符的用法，它以 3 种不同的计数系统显示了一个整数的值极其平方。注意，可以单独使用控制符，也可将其作为一系列插入的组成部分。

程序清单 17.3　manip.cpp

```
// manip.cpp -- using format manipulators
#include <iostream>
int main()
{
    using namespace std;
    cout << "Enter an integer: ";
    int n;
    cin >> n;

    cout << "n     n*n\n";
    cout << n << "     " << n * n << " (decimal)\n";
// set to hex mode
    cout << hex;
    cout << n << "     ";
    cout << n * n << " (hexadecimal)\n";

// set to octal mode
    cout << oct << n << "     " << n * n << " (octal)\n";

// alternative way to call a manipulator
    dec(cout);
    cout << n << "     " << n * n << " (decimal)\n";

    return 0;
}
```

下面程序清单 17.3 中程序的运行情况：

```
Enter an integer: 13
n     n*n
13     169 (decimal)
d     a9 (hexadecimal)
15     251 (octal)
13     169 (decimal)
```

2. 调整字段宽度

您可能已经注意到，在程序清单 17.3 的输出中各列并没有对齐，这是因为数字的字段宽度不相同。可以使用 width 成员函数将长度不同的数字放到宽度相同的字段中，该方法的原型为：

```
int width();
int width(int i);
```

第一种格式返回字段宽度的当前设置；第二种格式将字段宽度设置为 i 个空格，并返回以前的字段宽度值。这使得能够保存以前的值，以便以后恢复宽度值时使用。

width( )方法只影响将显示的下一个项目，然后字段宽度将恢复为默认值。例如，请看下面的语句：

```
cout << '#';
cout.width(12);
cout << 12 << "#" <<  24 << "#\n";
```

由于 width( )是成员函数，因此必须使用对象（这里为 cout）来调用它。输出语句生成的输出如下：

```
#           12#24#
```

12 被放到宽度为 12 个字符的字段的最右边，这被称为右对齐。然后，字段宽度恢复为默认值，并将两个#符号以及 24 放在宽度与它们的长度相等的字段中。

警告：width( )方法只影响接下来显示的一个项目，然后字段宽度将恢复为默认值。

C++永远不会截短数据，因此如果试图在宽度为 2 的字段中打印一个 7 位值，C++将增宽字段，以容纳该数据（在有些语言中，如果数据长度与字段宽度不匹配，将用星号填充字段。C/C++的原则是：显示所有的数据比保持列的整洁更重要。C++视内容重于形式）。程序清单 17.4 演示了 width( )成员函数是如何工作的。

程序清单 17.4　width.cpp

```
// width.cpp -- using the width method
#include <iostream>

int main()
{
    using std::cout;
    int w = cout.width(30);
    cout << "default field width = " << w << ":\n";
    cout.width(5);
    cout << "N" <<':';
    cout.width(8);
    cout << "N * N" << ":\n";

    for (long i = 1; i <= 100; i *= 10)
    {
        cout.width(5);
        cout << i <<':';
        cout.width(8);
        cout << i * i << ":\n";
    }

    return 0;
}
```

程序清单 17.4 中程序的输出如下：

```
      default field width = 0:
    N:   N * N:
    1:       1:
   10:     100:
  100:   10000:
```

在上述输出中，值在字段中右对齐。输出中包含空格，也就是说，cout 通过加入空格来填满整个字段。右对齐时，空格被插入到值的左侧。用来填充的字符叫做填充字符（fill character）。右对齐是默认的。

注意，在程序清单 17.4 中，第一条 cout 语句显示字符串时，字段宽度被设置为 30，但在显示 w 的值时，字段宽度不是 30。这是由于 width( )方法只影响接下来被显示的一个项目。另外，w 的值为 0。这是由于 cout.width（30）返回的是以前的字段宽度，而不是刚设置的值。W 为 0 表明，默认的字段宽度为 0。由

于 C++总会增长字段，以容纳数据，因此这种值适用于所有的数据。最后，程序使用 width( )来对齐列标题和数据，方法是将第 1 列宽度设置为 5 个字符，将第 2 列的宽度设置为 8 个字符。

3. 填充字符

在默认情况下，cout 用空格填充字段中未被使用的部分，可以用 fill( )成员函数来改变填充字符。例如，下面的函数调用将填充字符改为星号：

```
cout.fill('*');
```

这对于检查打印结果，防止接收方添加数字很有用。程序清单 17.5 演示了该成员函数的用法。

程序清单 17.5　fill.cpp

```
// fill.cpp -- changing fill character for fields
#include <iostream>

int main()
{
    using std::cout;
    cout.fill('*');
    const char * staff[2] = { "Waldo Whipsnade", "Wilmarie Wooper"};
    long bonus[2] = {900, 1350};

    for (int i = 0; i < 2; i++)
    {
        cout << staff[i] << ": $";
        cout.width(7);
        cout << bonus[i] << "\n";
    }

    return 0;
}
```

下面是程序清单 17.5 中程序的输出：

```
Waldo Whipsnade: $****900
Wilmarie Wooper: $***1350
```

注意，与字段宽度不同的是，新的填充字符将一直有效，直到更改它为止。

4. 设置浮点数的显示精度

浮点数精度的含义取决于输出模式。在默认模式下，它指的是显示的总位数。在定点模式和科学模式下（稍后将讨论），精度指的是小数点后面的位数。已经知道，C++的默认精度为 6 位（但末尾的 0 将不显示）。precision( )成员函数使得能够选择其他值。例如，下面语句将 cout 的精度设置为 2：

```
cout.precision(2);
```

和 width( )的情况不同，但与 fill( )类似，新的精度设置将一直有效，直到被重新设置。程序清单 17.6 准确地说明了这一点。

程序清单 17.6　precise.cpp

```
// precise.cpp -- setting the precision
#include <iostream>

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

```
    float price1 = 20.40;
    float price2 = 1.9 + 8.0 / 9.0;

    cout << "\"Furry Friends\" is $" << price1 << "!\n";
    cout << "\"Fiery Fiends\" is $" << price2 << "!\n";

    cout.precision(2);
    cout << "\"Furry Friends\" is $" << price1 << "!\n";
    cout << "\"Fiery Fiends\" is $" << price2 << "!\n";

    return 0;
}
```

下面是程序清单 17.6 中程序的输出：

```
"Furry Friends" is $20.4!
"Fiery Fiends" is $2.78889!
"Furry Friends" is $20!
"Fiery Fiends" is $2.8!
```

注意，第 3 行没有打印小数点及其后面的内容。另外，第 4 行显示的总位数为 2 位。

5. 打印末尾的 0 和小数点

对于有些输出（如价格或栏中的数字），保留末尾的 0 将更为美观。例如，对于程序清单 17.6 的输出，$20.40 将比$20.4 更美观。iostream 系列类没有提供专门用于完成这项任务的函数，但 ios_base 类提供了一个 setf( )函数（用于 set 标记），能够控制多种格式化特性。这个类还定义了多个常量，可用作该函数的参数。例如，下面的函数调用使 cout 显示末尾小数点：

```
cout.setf(ios_base::showpoint);
```

使用默认的浮点格式时，上述语句还将导致末尾的 0 被显示出来。也就是说，如果使用默认精度（6 位）时，cout 不会将 2.00 显示为 2，而是将它显示为 2.000000。程序清单 17.7 在程序清单 17.6 中添加了这条语句。

您可能对表示法 ios_base::showpoint 有疑问，showpoint 是 ios_base 类声明中定义的类级静态常量。类级意味着如果在成员函数定义的外面使用它，则必须在常量名前面加上作用域运算符（::）。因此 ios_base::showpoint 指的是在 ios_base 类中定义的一个常量。

程序清单 17.7　showpt.cpp

```
// showpt.cpp -- setting the precision, showing trailing point
#include <iostream>

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

    float price1 = 20.40;
    float price2 = 1.9 + 8.0 / 9.0;

    cout.setf(ios_base::showpoint);
    cout << "\"Furry Friends\" is $" << price1 << "!\n";
    cout << "\"Fiery Fiends\" is $" << price2 << "!\n";

    cout.precision(2);
    cout << "\"Furry Friends\" is $" << price1 << "!\n";
```

```
    cout << "\"Fiery Fiends\" is $" << price2 << "!\n";

    return 0;
}
```

下面是使用当前 C++格式时，程序清单 17.7 中程序的输出：

```
"Furry Friends" is $20.4000!
"Fiery Fiends" is $2.78889!
"Furry Friends" is $20.!
"Fiery Fiends" is $2.8!
```

在上述输出中，第一行显示了；第三行显示了小数点，但没有显示末尾的 0，这是因为精度被设置为 2，而小数点前面已经包含两位。

6. 再谈 setf( )

setf( )方法控制了小数点被显示时其他几个格式选项，因此来仔细研究一下它。ios_base 类有一个受保护的数据成员，其中的各位（这里叫作标记）分别控制着格式化的各个方面，例如计数系统、是否显示末尾的 0 等。打开一个标记称为设置标记（或位），并意味着相应的位被设置为 1。位标记是编程开关，相当于设置 DIP 开关以配置计算机硬件。例如，hex、dec 和 oct 控制符调整控制计数系统的 3 个标记位。setf( )函数提供了另一种调整标记位的途径。

setf( )函数有两个原型。第一个为：

```
fmtflags setf(fmtflags);
```

其中，fmtflags 是 bitmask 类型（参见后面的“注意”）的 typedef 名，用于存储格式标记。该名称是在 ios_base 类中定义的。这个版本的 setf( )用来设置单个位控制的格式信息。参数是一个 fmtflags 值，指出要设置哪一位。返回值是类型为 fmtflags 的数字，指出所有标记以前的设置。如果打算以后恢复原始设置，则可以保存这个值。应给 setf( )传递什么呢？如果要第 11 位设置为 1，则可以传递一个第 11 位为 1 的数字。返回值的第 11 位将被设置为 1。对位进行跟踪好像单调乏味（实际上也是这样）。然而，您不必作做这项工作，ios_base 类定义了代表位值的常量，表 17.1 列出了其中的一些定义。

表 17.1　　格式常量

常　　量	含　　义
ios_base ::boolalpha	输入和输出 bool 值，可以为 true 或 false
ios_base ::showbase	对于输出，使用 C++基数前缀（0，0x）
ios_base ::showpoint	显示末尾的小数点
ios_base ::uppercase	对于 16 进制输出，使用大写字母，E 表示法
ios_base ::showpos	在正数前面加上+

注意：bitmask 类型是一种用来存储各个位值的类型。它可以是整型、枚举，也可以是 STL bitset 容器。这里的主要思想是，每一位都是可以单独访问的，都有自己的含义。iostream 软件包使用 bitmask 来存储状态信息。

由于这些格式常量都是在 ios_base 类中定义，因此使用它们时，必须加上作用域解析运算符。也就是说，应使用 ios_base ::uppercase，而不是 uppercase。如果不想使用 using 编译指令或 using 声明，可以使用作用域运算符来指出这些名称位于名称空间 std 中。修改将一直有效，直到被覆盖为止。程序清单 17.8 演示了如何使用其中一些常量。

程序清单 17.8　setf.cpp

```
// setf.cpp -- using setf() to control formatting
#include <iostream>
```

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

    int temperature = 63;
    cout << "Today's water temperature: ";
    cout.setf(ios_base::showpos);    // show plus sign
    cout << temperature << endl;

    cout << "For our programming friends, that's\n";
    cout << std::hex << temperature << endl; // use hex
    cout.setf(ios_base::uppercase);   // use uppercase in hex
    cout.setf(ios_base::showbase);    // use 0X prefix for hex
    cout << "or\n";
    cout << temperature << endl;
    cout << "How " << true << "!  oops -- How ";
    cout.setf(ios_base::boolalpha);
    cout << true << "!\n";

    return 0;
}
```

下面是程序清单 17.8 中程序的输出：

```
Today's water temperature: +63
For our programming friends, that's
3f
or
0X3F
How 0X1!  oops -- How true!
```

注意，仅当基数为 10 时才使用加号。C++将十六进制和八进制都视为无符号的，因此对它们，无需使用符号（然而，有些 C++实现可能仍然会显示加号）。

第二个 setf( )原型接受两个参数，并返回以前的设置：

```
fmtflags setf(fmtflags , fmtflags );
```

函数的这种重载格式用于设置由多位控制的格式选项。第一参数和以前一样，也是一个包含了所需设置的 fmtflags 值。第二参数指出要清除第一个参数中的哪些位。例如，将第 3 位设置为 1 表示以 10 为基数，将第 4 位设置为 1 表示以 8 为基数，将第 5 位设置为 1 表示以 16 为基数。假设输出是以 10 为基数的，而要将它设置为以 16 为基数，则不仅需要将第 5 位设置为 1，还需要将第 3 位设置为 0——这叫作清除位（clearing the bit）。聪明的十六进制控制符可自动完成这两项任务。使用函数 setf( )时，要做的工作多些，因为要用第二参数指出要清除哪些位，用第一参数指出要设置哪位。这并不像听上去那么复杂，因为 ios_base 类为此定义了常量（如表 17.2 所示）。具体地说，要修改基数，可以将常量 ios_base::basefield 用作第二参数，将 ios_base ::hex 用作第一参数。也就是说，下面的函数调用与使用十六进制控制符的作用相同：

```
cout.setf(ios_base::hex, ios_base::basefield);
```

表 17.2　　setf(long, long)的参数

第二个参数	第一个参数	含　义
ios_base ::basefield	ios_base ::dec	使用基数 10
	ios_base ::oct	使用基数 8
	ios_base ::hex	使用基数 16

续表

第二个参数	第一个参数	含　义
ios_base ::floatfield	ios_base ::fixed	使用定点计数法
	ios_base ::scientific	使用科学计数法
ios_base ::adjustfield	ios_base ::left	使用左对齐
	ios_base ::right	使用右对齐
	ios_base ::internal	符号或基数前缀左对齐，值右对齐

ios_base 类定义了可按这种方式处理的 3 组格式标记。每组标记都由一个可用作第二参数的常量和两三个可用作第一参数的常量组成。第二参数清除一批相关的位，然后第一参数将其中一位设置为 1。表 17.2 列出了用作 setf( )的第二参数的常量的名称、可用作第一参数的相关常量以及它们的含义。例如，要选择左对齐，可将 ios_base ::adjustfield 用作第二参数，将 ios_base ::left 作为第一参数。左对齐意味着将值放在字段的左端，右对齐则表示将值放在字段的右端。内部对齐表示将符号或基数前缀放在字段左侧，余下的数字放在字段的右侧（遗憾的是，C++没有提供自对齐模式）。

定点表示法意味着使用格式 123.4 来表示浮点值，而不管数字的长度如何，科学表示法则意味着使用格式 1.23e04，而不考虑数字的长度。如果您熟悉 C 语言中 printf( )的说明符，则可能知道，默认的 C++模式对应于%g 说明符，定点表示法对应于%f 说明符，而科学表示法对应于%e 说明符。

在 C++标准中，定点表示法和科学表示法都有下面两个特征：

- 精度指的是小数位数，而不是总位数；
- 显示末尾的 0。

setf( )函数是 ios_base 类的一个成员函数。由于这个类是 ostream 类的基类，因此可以使用 cout 对象来调用该函数。例如，要左对齐，可使用下面的调用：

```
ios_base::fmtflags old = cout.setf(ios::left, ios::adjustfield);
```

要恢复以前的设置，可以这样做：

```
cout.setf(old, ios::adjustfield);
```

程序清单 17.9 是一个使用两个参数的 setf( )的示例。

注意：程序清单 17.9 中的程序使用了一个数学函数，有些 C++系统不自动搜索数学库。例如，有些 UNIX 系统要求这样做：

```
$ CC setf2.C -lm
```

-lm 选项命令链接程序搜索数学库。同样，有些使用 g++的 Linux 系统也要求这样做。

程序清单 17.9　setf2.cpp

```
// setf2.cpp -- using setf() with 2 arguments to control formatting
#include <iostream>
#include <cmath>

int main()
{
    using namespace std;
   // use left justification, show the plus sign, show trailing
    // zeros, with a precision of 3
    cout.setf(ios_base::left, ios_base::adjustfield);
    cout.setf(ios_base::showpos);
    cout.setf(ios_base::showpoint);
    cout.precision(3);
    // use e-notation and save old format setting
```

```
    ios_base::fmtflags old = cout.setf(ios_base::scientific,
        ios_base::floatfield);
    cout << "Left Justification:\n";
    long n;
    for (n = 1; n <= 41; n+= 10)
    {
        cout.width(4);
        cout << n << "|";
        cout.width(12);
        cout << sqrt(double(n)) << "|\n";
    }

    // change to internal justification
    cout.setf(ios_base::internal, ios_base::adjustfield);
    // restore default floating-point display style
    cout.setf(old, ios_base::floatfield);

    cout << "Internal Justification:\n";
    for (n = 1; n <= 41; n+= 10)
    {
        cout.width(4);
        cout << n << "|";
        cout.width(12);
        cout << sqrt(double(n)) << "|\n";
    }

    // use right justification, fixed notation
    cout.setf(ios_base::right, ios_base::adjustfield);
    cout.setf(ios_base::fixed, ios_base::floatfield);
    cout << "Right Justification:\n";
    for (n = 1; n <= 41; n+= 10)
    {
        cout.width(4);
        cout << n << "|";
        cout.width(12);
        cout << sqrt(double(n)) << "|\n";
    }

    return 0;
}
```

下面是程序清单 17.9 中程序的输出：

```
Left Justification:
+1  |+1.000e+00  |
+11 |+3.317e+00  |
+21 |+4.583e+00  |
+31 |+5.568e+00  |
+41 |+6.403e+00  |
Internal Justification:
+  1|+       1.00|
+ 11|+       3.32|
+ 21|+       4.58|
+ 31|+       5.57|
+ 41|+       6.40|
```

```
Right Justification:
  +1|       +1.000|
 +11|       +3.317|
 +21|       +4.583|
 +31|       +5.568|
 +41|       +6.403|
```

注意到精度 3 让默认的浮点显示（在这个程序中用于内部对齐）总共显示 3 位，而定点模式和科学模式只显示 3 位小数（e 表示法的指数位数取决于实现）。

调用 setf( )的效果可以通过 unsetf( )消除，后者的原型如下：

```
void unsetf(fmtflags mask);
```

其中，mask 是位模式。mask 中所有的位都设置为 1，将使得对应的位被复位。也就是说，setf( )将位设置为 1，unsetf( )将位恢复为 0。例如：

```
cout.setf(ios_base::showpoint);        // show trailing decimal point
cout.unsetf(ios_base::boolshowpoint); // don't show trailing decimal point
cout.setf(ios_base::boolalpha);        // display true, false
cout.unsetf(ios_base::boolalpha);      // display 1, 0
```

您可能注意到了，没有专门指示浮点数默认显示模式的标记。系统的工作原理如下：仅当只有定点位被设置时使用定点表示法；仅当只有科学位被设置时使用科学表示法；对于其他组合，如没有位被设置或两位都被设置时，将使用默认模式。因此，启用默认模式的方法之一如下：

```
cout.setf(0, ios_base::floatfield);  // go to default mode
```

第二个参数关闭这两位，而第一个参数不设置任何位。一种实现同样目标的简捷方式是，使用参数 ios::floatfield 来调用函数 unsetf( )：

```
cout.unsetf(ios_base::floatfield);   // go to default mode
```

如果已知 cout 处于定点状态，则可以使用参数 ios_base::fixed 调用函数 unsetf( )来切换到默认模式；然而，无论 cout 的当前状态如何，使用参数 ios_base::floatfield 调用函数 unsetf( )都将切换到默认模式，因此这是一种更好的选择。

7. 标准控制符

使用 setf( )不是进行格式化的、对用户最为友好的方法，C++提供了多个控制符，能够调用 setf( )，并自动提供正确的参数。前面已经介绍过 dec、hex 和 oct，这些控制符（多数都不适用于老式 C++实现）的工作方式都与 hex 相似。例如，下面的语句打开左对齐和定点选项：

```
cout << left << fixed;
```

表 17.3 列出了这些控制符以及其他一些控制符。

表 17.3　一些标准控制符

控制符	调用
boolalpha	setf(ios_base::boolalpha)
noboolalpha	unset(ios_base::noboolalpha)
showbase	setf(ios_base::showbase)
noshowbase	unsetf(ios_base::showbase)
showpoint	setf(ios_base::showpoint)
noshowpoint	unsetf(ios_base::showpoint)
showpos	setf(ios_base::showpos)
noshowpos	unsetf(ios_base::showpos)
uppercase	setf(ios_base::uppercase)

续表

控制符	调用
nouppercase	unsetf(ios_base::uppercase)
internal	setf(ios_base::internal, ios_base::adjustfield)
left	setf(ios_base::left, ios_base::adjustfield)
right	setf(ios_base::right, ios_base::adjustfield)
dec	setf(ios_base::dec, ios_base::basefield)
hex	setf(ios_base::hex, ios_base::basefield)
oct	setf(ios_base::oct, ios_base::basefield)
fixed	setf(ios_base::fixed, ios_base::floatfield)
scientific	setf(ios_base::scientific, ios_base::floatfield)

提示：如果系统支持这些控制符，请使用它们；否则，仍然可以使用 setf( )。

8. 头文件 iomanip

使用 iostream 工具来设置一些格式值（如字段宽度）不太方便。为简化工作，C++在头文件 iomanip 中提供了其他一些控制符，它们能够提供前面讨论过的服务，但表示起来更方便。3 个最常用的控制符分别是 setprecision( )、setfill( )和 setw( )，它们分别用来设置精度、填充字符和字段宽度。与前面讨论的控制符不同的是，这 3 个控制符带参数。setprecision( )控制符接受一个指定精度的整数参数；setfill( ) 控制符接受一个指定填充字符的 char 参数；setw( )控制符接受一个指定字段宽度的整数参数。由于它们都是控制符，因此可以用 cout 语句连接起来。这样，setw( )控制符在显示多列值时尤其方便。程序清单 17.10 演示了这一点，它对于每一行输出，都多次修改了字段宽度和填充字符，同时使用了一些较新的标准控制符。

注意：有些 C++系统不自动搜索数学库。前面说过，有些 UNIX 系统要求使用如下命令选项来访问数学库：

```
$ CC iomanip.C -lm
```

程序清单 17.10　iomanip.cpp

```
// iomanip.cpp -- using manipulators from iomanip
// some systems require explicitly linking the math library
#include <iostream>
#include <iomanip>
#include <cmath>

int main()
{
    using namespace std;
    // use new standard manipulators
    cout << fixed << right;

    // use iomanip manipulators
    cout << setw(6) << "N" << setw(14) << "square root"
         << setw(15) << "fourth root\n";
```

```
    double root;
    for (int n = 10; n <=100; n += 10)
    {
        root = sqrt(double(n));
        cout << setw(6) << setfill('.') << n << setfill(' ')
               << setw(12) << setprecision(3) << root
               << setw(14) << setprecision(4) << sqrt(root)
               << endl;
    }

    return 0;
}
```

下面是程序清单 17.10 中程序的输出：

```
     N   square root   fourth root
....10       3.162        1.7783
....20       4.472        2.1147
....30       5.477        2.3403
....40       6.325        2.5149
....50       7.071        2.6591
....60       7.746        2.7832
....70       8.367        2.8925
....80       8.944        2.9907
....90       9.487        3.0801
...100      10.000        3.1623
```

现在可以生成几乎完全对齐的列了。使用 fixed 控制符导致显示末尾的 0。

17.3　使用 cin 进行输入

现在来介绍输入，即如何给程序提供数据。cin 对象将标准输入表示为字节流。通常情况下，通过键盘来生成这种字符流。如果键入字符序列 2011，cin 对象将从输入流中抽取这几个字符。输入可以是字符串的一部分、int 值、float 值，也可以是其他类型。因此，抽取还涉及了类型转换。cin 对象根据接收值的变量的类型，使用其方法将字符序列转换为所需的类型。

通常，可以这样使用 cin：

```
cin >> value_holder;
```

其中，value_holder 为存储输入的内存单元，它可以是变量、引用、被解除引用的指针，也可以是类或结构的成员。cin 解释输入的方式取决于 value_holder 的数据类型。istream 类（在 iostream 头文件中定义）重载了抽取运算符>>，使之能够识别下面这些基本类型：

- signed char &；
- unsigned char &；
- char &；
- short &；
- unsigned short &；
- int &；
- unsigned int &；
- long &；
- unsigned long &；
- long long &（C++11）；

- unsigned long long &（C++11）；
- float &；
- double &；
- long double &。

这些运算符函数被称为格式化输入函数（formatted input functions），因为它们可以将输入数据转换为目标指定的格式。

典型的运算符函数的原型如下：

```
istream & operator>>(int &);
```

参数和返回值都是引用。引用参数（参见第 8 章）意味着下面这样的语句将导致 operator>>( )函数处理变量 staff_size 本身，而不是像常规参数那样处理它的副本：

```
cin >> staff_size;
```

由于参数类型为引用，因此 cin 能够直接修改用作参数的变量的值。例如，上述语句将直接修改变量 staff_size 的值。稍后将介绍引用返回值的重要意义。首先来看抽取运算符的类型转换方面。对于上述列出的各种类型的参数，抽取运算符将字符输入转换为指定类型的值。例如，假设 staff_size 的类型为 int，则编译器将：

```
cin >> staff_size;
```

与下面的原型匹配：

```
istream & operator>>(int &);
```

对应于上述原型的函数将读取发送给程序的字符流（假设为字符 2、3、1、8 和 4）。对于使用 2 字节 int 的系统来说，函数将把这些字符转换为整数 23184 的 2 字节二进制表示。如果 staff_size 的类型为 double，则 cin 将使用 operator >> (double &)将上述输入转换为值 23184.0 的 8 字节浮点表示。

顺便说一句，可以将 hex、oct 和 dec 控制符与 cin 一起使用，来指定将整数输入解释为十六进制、八进制还是十进制格式。例如，下面的语句将输入 12 或 0x12 解释为十六进制的 12 或十进制的 18，而将 ff 或 FF 解释为十进制的 255：

```
cin >> hex;
```

istream 类还为下列字符指针类型重载了>>抽取运算符：

- signed char *；
- char *；
- unsigned char *。

对于这种类型的参数，抽取运算符将读取输入中的下一个单词，将它放置到指定的地址，并加上一个空值字符，使之成为一个字符串。例如，假设有这样一段代码：

```
cout << "Enter your first name:\n";
char name[20];
cin >> name;
```

如果通过键入 Liz 来进行响应，则抽取运算符将把字符 Liz\0 放到 name 数组中（\0 表示末尾的空值字符）。name 标识符是一个 char 数组名，可作为数组第一个元素的地址，这使 name 的类型为 char *（指向 char 的指针）。

每个抽取运算符都返回调用对象的引用，这使得能够将输入拼接起来，就像拼接输出那样：

```
char name[20];
float fee;
int group;
cin >> name >> fee >> group;
```

其中，cin>>name 返回的 cin 对象成了处理 fee 的对象。

17.3.1 cin>>如何检查输入

不同版本的抽取运算符查看输入流的方法是相同的。它们跳过空白（空格、换行符和制表符），直到遇

到非空白字符。即使对于单字符模式（参数类型为 char、unsigned char 或 signed char），情况也是如此，但对于 C 语言的字符输入函数，情况并非如此（参见图 17.5）。在单字符模式下，>>运算符将读取该字符，将它放置到指定的位置。在其他模式下，>>运算符将读取一个指定类型的数据。也就是说，它读取从非空白字符开始，到与目标类型不匹配的第一个字符之间的全部内容。

```
char philosophy[20];
int distance;
char initial;

cin >> philosophy >> distance >> initial;
```

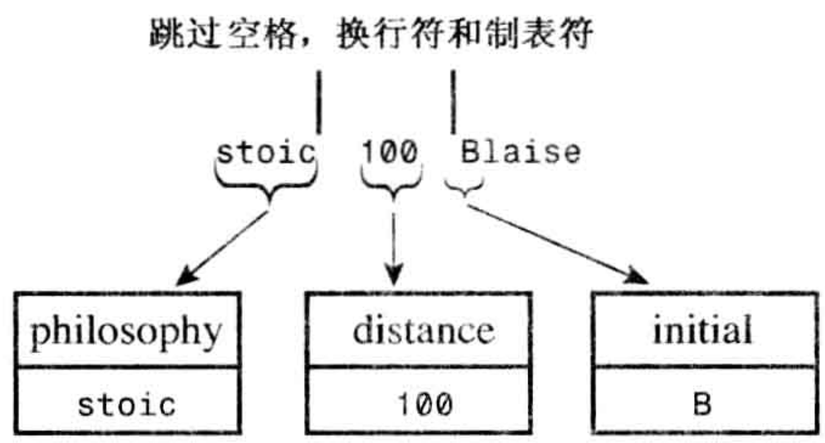

图 17.5　cin>>跳过空白

例如，对于下面的代码：

```
int elevation;
cin >> elevation;
```

假设键入下面的字符：

```
-123Z
```

运算符将读取字符-、1、2 和 3，因为它们都是整数的有效部分。但 Z 字符不是有效字符，因此输入中最后一个可接受的字符是 3。Z 将留在输入流中，下一个 cin 语句将从这里开始读取。与此同时，运算符将字符序列-123 转换为一个整数值，并将它赋给 elevation。

输入有时可能没有满足程序的期望。例如，假设输入的是 Zcar，而不是-123Z。在这种情况下，抽取运算符将不会修改 elevation 的值，并返回 0（如果 istream 对象的错误状态被设置，if 或 while 语句将判定该对象为 false，这将在本章后面做更详细的介绍）。返回值 false 让程序能够检查输入是否满足要求，如程序清单 17.11 所示。

程序清单 17.11　check_it.cpp

```
// check_it.cpp -- checking for valid input
#include <iostream>

int main()
{
    using namespace std;
    cout << "Enter numbers: ";

    int sum = 0;
    int input;
    while (cin >> input)
    {
        sum += input;
    }

    cout << "Last value entered = " << input << endl;
    cout << "Sum = " << sum << endl;
    return 0;
}
```

下面是输入流中包含不适当输入（-123Z）时程序清单 17.11 中程序的输出：

```
Enter numbers: 200
10 -50 -123Z 60
Last value entered = -123
Sum = 37
```

由于输入被缓冲。因此通过键盘输入的第二行在用户按下回车键之前，不会被发送给程序。然而，循环在字符 Z 处停止了对输入的处理，因此它不与任何一种浮点格式匹配。输入与预期格式不匹配反过来将导致表达式 cin>>input 的结果为 false，因此 while 循环被终止。

17.3.2 流状态

我们来进一步看看不适当的输入会造成什么后果。cin 或 cout 对象包含一个描述流状态（stream state）的数据成员（从 ios_base 类那里继承的）。流状态（被定义为 iostate 类型，而 iostate 是一种 bitmask 类型）由 3 个 ios_base 元素组成：eofbit、badbit 或 failbit，其中每个元素都是一位，可以是 1（设置）或 0（清除）。当 cin 操作到达文件末尾时，它将设置 eofbit；当 cin 操作未能读取到预期的字符时（像前一个例子那样），它将设置 failbit。I/O 失败（如试图读取不可访问的文件或试图写入写保护的磁盘），也可能将 failbit 设置为 1。在一些无法诊断的失败破坏流时，badbit 元素将被设置（实现没有必要就哪些情况下设置 failbit，哪些情况下设置 badbit 达成一致）。当全部 3 个状态位都设置为 0 时，说明一切顺利。程序可以检查流状态，并使用这种信息来决定下一步做什么。表 17.4 列出了这些位和一些报告或改变流状态的 ios_base 方法。

表 17.4　　流状态

成　员	描　述
eofbit	如果到达文件尾，则设置为 1
badbit	如果流被破坏，则设置为 1；例如，文件读取错误
failbit	如果输入操作未能读取预期的字符或输出操作没有写入预期的字符，则设置为 1
goodbit	另一种表示 0 的方法
good( )	如果流可以使用（所有的位都被清除），则返回 true
eof( )	如果 eofbit 被设置，则返回 true
bad( )	如果 badbit 被设置，则返回 true
fail( )	如果 badbit 或 failbit 被设置，则返回 true
rdstate( )	返回流状态
exceptions( )	返回一个位掩码，指出哪些标记导致异常被引发
exceptions(isostate ex)	设置哪些状态将导致 clear( )引发异常；例如，如果 ex 是 eofbit，则如果 eofbit 被设置，clear( )将引发异常
clear(iostate s)	将流状态设置为 s；s 的默认值为 0（goodbit）；如果(restate( )& exceptions( ))! = 0，则引发异常 basic_ios::failure
setstate(iostate s)	调用 clear（rdstate( )| s）。这将设置与 s 中设置的位对应的流状态位，其他流状态位保持不变

1. 设置状态

表 17.4 中的两种方法——clear( )和 setstate( )很相似。它们都重置状态，但采取的方式不同。clear( )方法将状态设置为它的参数。因此，下面的调用将使用默认参数 0，这将清除全部 3 个状态位（eofbit、badbit 和 failbit）：

```
clear();
```

同样，下面的调用将状态设置为 eofbit；也就是说，eofbit 将被设置，另外两个状态位被清除：

```
clear(eofbit);
```

而 setstate( )方法只影响其参数中已设置的位。因此，下面的调用将设置 eofbit，而不会影响其他位：

```
setstate(eofbit);
```

因此，如果 failbit 被设置，则仍将被设置。

为什么需要重新设置流状态呢？对于程序员来说，最常见的理由是，在输入不匹配或到达文件尾时，需要使用不带参数的 clear( )重新打开输入。这样做是否有意义，取决于程序要执行的任务。稍后将介绍一些例子。setstate( )的主要用途是为输入和输出函数提供一种修改状态的途径。例如，如果 num 是一个 int，则下面的调用将可能导致 operator >> (int &)使用 setstate( )设置 failbit 或 eofbit：

```
cin >> num;  // read an int
```

2. I/O 和异常

假设某个输入函数设置了 eofbit，这是否会导致异常被引发呢？在默认情况下，答案是否定的。但可以使用 exceptions( )方法来控制异常如何被处理。

首先，介绍一些背景知识。exceptions( )方法返回一个位字段，它包含 3 位，分别对应于 eofbit、failbit 和 badbit。修改流状态涉及 clear( )或 setstate( )，这都将使用 clear( )。修改流状态后，clear( )方法将当前的流状态与 exceptions( )返回的值进行比较。如果在返回值中某一位被设置，而当前状态中的对应位也被设置，则 clear( )将引发 ios_base::failure 异常。如果两个值都设置了 badbit，将发生这种情况。如果 exceptions( )返回 goodbit，则不会引发任何异常。ios_base::failure 异常类是从 std::exception 类派生而来的，因此包含一个 what( )方法。

exceptions( )的默认设置为 goodbit，也就是说，没有引发异常。但重载的 exceptions（iostate）函数使得能够控制其行为：

```
cin.exceptions(badbit);  // setting badbit causes exception to be thrown
```

位运算符 OR（在附录 E 讨论）使得能够指定多位。例如，如果 badbit 或 eofbit 随后被设置，下面的语句将引发异常：

```
cin.exceptions(badbit | eofbit);
```

程序清单 17.12 对程序清单 17.11 进行了修改，以便程序能够在 failbit 被设置时引发并捕获异常。

程序清单 17.12　cinexcp.cpp

```
// cinexcp.cpp -- having cin throw an exception
#include <iostream>
#include <exception>

int main()
{
    using namespace std;
    // have failbit cause an exception to be thrown
    cin.exceptions(ios_base::failbit);
    cout << "Enter numbers: ";
    int sum = 0;
    int input;
    try {
        while (cin >> input)
        {
            sum += input;
        }
    } catch(ios_base::failure & bf)
```

```
    {
        cout << bf.what() << endl;
        cout << "O! the horror!\n";
    }

    cout << "Last value entered = " << input << endl;
    cout << "Sum = " << sum << endl;
    return 0;
}
```

程序清单 17.12 中程序的运行情况如下，其中的 what( )消息取决于实现：

```
Enter numbers: 20 30 40 pi 6
ios_base failure in clear
O! the horror!
Last value entered = 40.00
Sum = 90.00
```

这就是如何在接受输入时使用异常。然而，应该使用它们吗？这取决于具体情况。就这个例子而言，答案是否定的。异常用于捕获不正常的意外情况，但这个例子将输入错误作为一种退出循环的方式。然而，让这个程序在 badbit 位被设置时引发异常可能是合理的，因为这种情况是意外的。如果程序被设计成从一个数据文件中读取数据，直到到达文件尾，则在 failbit 位被设置时引发异常也是合理的，因为这表明数据文件出现了问题。

3. 流状态的影响

只有在流状态良好（所有的位都被清除）的情况下，下面的测试才返回 true：

```
while (cin >> input)
```

如果测试失败，可以使用表 17.4 中的成员函数来判断可能的原因。例如，可以将程序清单 17.11 中的核心部分修改成这样：

```
while (cin >> input)
{
    sum += input;
}
if (cin.eof())
    cout << "Loop terminated because EOF encountered\n";
```

设置流状态位有一个非常重要的后果：流将对后面的输入或输出关闭，直到位被清除。例如，下面的代码不可行：

```
while (cin >> input)
{
    sum += input;
}
cout << "Last value entered = " << input << endl;
cout << "Sum = " << sum << endl;
cout << "Now enter a new number: ";
cin >> input;   // won't work
```

如果希望程序在流状态位被设置后能够读取后面的输入，就必须将流状态重置为良好。这可以通过调用 clear( )方法来实现：

```
while (cin >> input)
{
    sum += input;
}
cout << "Last value entered = " << input << endl;
```

```
cout << "Sum = " << sum << endl;
cout << "Now enter a new number: ";
cin.clear();          // reset stream state
while (!isspace(cin.get()))
    continue;         // get rid of bad input
cin >> input;         // will work now
```

注意，这还不足以重新设置流状态。导致输入循环终止的不匹配输入仍留在输入队列中，程序必须跳过它。一种方法是一直读取字符，直到到达空白为止。isspace( )函数（参见第 6 章）是一个 cctype 函数，它在参数是空白字符时返回 true。另一种方法是，丢弃行中的剩余部分，而不仅仅是下一个单词：

```
while (cin.get() != '\n')
    continue;  // get rid rest of line
```

这个例子假设循环由于不恰当的输入而终止。现在，假设循环是由于到达文件尾或者由于硬件故障而终止的，则处理错误输入的新代码将毫无意义。可以使用 fail( )方法检测假设是否正确，来修复问题。由于历史原因，fail( )在 failbit 或 eofbit 被设置时返回 true，因此代码必须排除后一种情况。下面是一个排除这种情况的例子：

```
while (cin >> input)
{
    sum += input;
}
cout << "Last value entered = " << input << endl;
cout << "Sum = " << sum << endl;
if (cin.fail() && !cin.eof() ) // failed because of mismatched input
{
      cin.clear();       // reset stream state
      while (!isspace(cin.get()))
           continue;     // get rid of bad input
}
else // else bail out
{
      cout << "I cannot go on!\n";
      exit(1);
}
cout << "Now enter a new number: ";
cin >> input;  // will work now
```

17.3.3　其他 istream 类方法

第 3 章～第 5 章讨论了 get( )和 getline( )方法。您可能还记得，它们提供下面的输入功能：

- 方法 get(char&)和 get(void)提供不跳过空白的单字符输入功能；
- 函数 get(char*, int, char)和 getline(char*, int, char)在默认情况下读取整行而不是一个单词。

它们被称为非格式化输入函数（unformatted input functions），因为它们只是读取字符输入，而不会跳过空白，也不进行数据转换。

来看一下 istream 类的这两组成员函数。

1. 单字符输入

在使用 char 参数或没有参数的情况下，get( )方法读取下一个输入字符，即使该字符是空格、制表符或换行符。get(char & ch)版本将输入字符赋给其参数，而 get(void)版本将输入字符转换为整型（通常是 int），并将其返回。

（1）成员函数 get(char &)

先来看 get(char &)。假设程序中包含如下循环：

```
int ct = 0;
char ch;
cin.get(ch);
while (ch != '\n')
{
    cout << ch;
    ct++;
    cin.get(ch);
}
cout << ct << endl;
```

接下来，假设提供了如下输入：

```
I C++ clearly.<Enter>
```

按下回车键后，这行输入将被发送给程序。上述程序片段将首先读取字符 I，使用 cout 显示它，并将 ct 递增到 1。接着，它读取 I 后面的空格字符，显示它，并将 ct 递增到 2。这一过程将一直继续下去，直到程序将回车键作为换行符处理，并终止循环。这里的重点是，通过使用 get(ch)，代码读取、显示并考虑空格和可打印字符。

假设程序试图使用>>：

```
int ct = 0;
char ch;
cin >> ch;
while (ch != '\n')     // FAILS
{
    cout << ch;
    ct++;
    cin >> ch;
}
cout << ct << endl;
```

则代码将首先跳过空格，这样将不考虑空格，因此相应的输出压缩为如下：

```
IC++clearly.
```

更糟糕的是，循环不会终止！由于抽取运算符跳过了换行符，因此代码不会将换行符赋给 ch，所以 while 循环测试将不会终止循环。

get(char &)成员函数返回一个指向用于调用它的 istream 对象的引用，这意味着可以拼接 get(char &)后面的其他抽取：

```
char c1, c2, c3;
cin.get(c1).get(c2) >> c3;
```

首先，cin.get(c1)将第一个输入字符赋给 c1，并返回调用对象——cin。这样代码缩为 cin.get(c2) >> c3，它将第二个输入字符赋给 c2。该函数调用返回 cin，将代码缩为 cin>>c3。这将把下一个非空白字符赋给 c3。因此 c1 和 c2 的值最后为空格，但 c3 不是。

如果 cin.get(char &)到达文件尾——无论是真正的文件尾，还是通过键盘仿真的文件尾（对于 DOS 和 Windows 命令提示符模式，为按下 Ctrl + Z；对于 UNIX，是在行首按下 Ctrl + D），它都不会给其参数赋值。这是完全正确的，因为如果程序到达文件尾，就没有值可供赋给参数了。另外，该方法还调用 setstate（failbit），导致 cin 的测试结果为 false：

```
char ch;
while (cin.get(ch))
{
    // process input
}
```

只要存在有效输入，cin.get(ch)的返回值都将是 cin，此时的判定结果为 true，因此循环将继续。到达

文件尾时，返回值判定为 false，循环终止。

（2）成员函数 get(void)

get(void)成员函数还读取空白，但使用返回值来将输入传递给程序。因此可以这样使用它：

```
int ct = 0;
char ch;
ch = cin.get();       // use return value
while (ch != '\n')
{
    cout << ch;
    ct++;
    ch = cin.get();
}
cout << ct << endl;
```

get(void)成员函数的返回类型为 int（或某种更大的整型，这取决于字符集和区域）。这使得下面的代码是非法的：

```
char c1, c2, c3;
cin.get().get() >> c3;  // not valid
```

这里，cin.get( )将返回一个 int 值。由于返回值不是类对象，因此不能对它应用成员运算符。因此将出现语法错误。然而，可以在抽取序列的最后使用 get( )：

```
char c1;
cin.get(c1).get();   // valid
```

get(void)的返回类型为 int，这意味着它后面不能跟抽取运算符。然而，由于 cin.get(c1)返回 cin，因此它可以放在 get( )的前面。上述代码将读取第一个输入字符，将其赋给 c1，然后读取并丢弃第二个输入字符。

到达文件尾后（不管是真正的文件尾还是模拟的文件尾），cin.get(void)都将返回值 EOF——头文件 iostream 提供的一个符号常量。这种设计特性使得可以这样来读取输入：

```
int ch;
while ((ch = cin.get()) != EOF)
{
    // process input
}
```

这里应将 ch 的类型声明为 int，而不是 char，因为值 EOF 可能无法使用 char 类型来表示。

第 5 章更详细地介绍了这些函数，表 17.5 对单字符输入函数的特性进行了总结。

表 17.5　　cin.get(ch)与 cin.get( )

特　征	cin.get(ch)	ch = cin.get( )
传输输入字符的方法	赋给参数 ch	将函数返回值赋给 ch
字符输入时函数的返回值	指向 istream 对象的引用	字符编码（int 值）
达到文件尾时函数的返回值	转换为 false	EOF

2. 采用哪种单字符输入形式

假设可以选择>>、get（char &）或 get（void），应使用哪一个呢？首先，应确定是否希望跳过空白。如果跳过空白更方便，则使用抽取运算符>>。例如，提供菜单选项时，跳过空白更为方便：

```
cout  << "a. annoy client          b. bill client\n"
      << "c. calm client           d. deceive client\n"
      << "q.\n";
```

```
cout  << "Enter a, b, c, d, or q: ";
char ch;
cin >> ch;
while (ch != 'q')
{
    switch(ch)
    {
        ...
    }
    cout << "Enter a, b, c, d, or q: ";
    cin >> ch;
}
```

要输入 b 进行响应，可以键入 b 并按回车键，这将生成两个字符的响应——b\n。如果使用 get( )，则必须添加在每次循环中处理\n 字符的代码，而抽取运算符可以跳过它（如果使用过 C 语言进行编程，则可能遇到过使用换行符进行响应为非法的情况。这是个很容易解决的问题，但比较讨厌）。

如果希望程序检查每个字符，请使用 get( )方法，例如，计算字数的程序可以使用空格来判断单词何时结束。在 get( )方法中，get(char &)的接口更佳。get(void)的主要优点是，它与标准 C 语言中的 getchar( )函数极其类似，这意味着可以通过包含 iostream（而不是 stdio.h），并用 cin.get( )替换所有的 getchar( )，用 cout.put(ch)替换所有的 putchar(ch)，来将 C 程序转换为 C++程序。

3. 字符串输入：getline( )、get( )和 ignore( )

接下来复习一下第 4 章介绍的字符串输入成员函数。getline( )成员函数和 get( )的字符串读取版本都读取字符串，它们的函数特征标相同（这是从更为通用的模板声明简化而来的）：

```
istream & get(char *, int, char);
istream & get(char *, int);
istream & getline(char *, int, char);
istream & getline(char *, int);
```

第一个参数是用于放置输入字符串的内存单元的地址。第二个参数比要读取的最大字符数大 1（额外的一个字符用于存储结尾的空字符，以便将输入存储为一个字符串）。第 3 个参数指定用作分界符的字符，只有两个参数的版本将换行符用作分界符。上述函数都在读取最大数目的字符或遇到换行符后为止。

例如，下面的代码将字符输入读取到字符数组 line 中：

```
char line[50];
cin.get(line, 50);
```

cin.get( )函数将在到达第 49 个字符或遇到换行符（默认情况）后停止将输入读取到数组中。get( )和 getline( )之间的主要区别在于，get( )将换行符留在输入流中，这样接下来的输入操作首先看到是将是换行符，而 gerline( )抽取并丢弃输入流中的换行符。

第 4 章演示了如何使用这两个成员函数的默认格式。现在来看一下接受三个参数的版本，第三个参数用于指定分界符。遇到分界字符后，输入将停止，即使还未读取最大数目的字符。因此，在默认情况下，如果在读取指定数目的字符之前到达行尾，这两种方法都将停止读取输入。和默认情况一样，get( )将分界字符留在输入队列中，而 getline( )不保留。

程序清单 17.13 演示了 getline( )和 get( )是如何工作的，它还介绍了 ignore( )成员函数。该函数接受两个参数：一个是数字，指定要读取的最大字符数；另一个是字符，用作输入分界符。例如，下面的函数调用读取并丢弃接下来的 255 个字符或直到到达第一个换行符：

```
cin.ignore(255, '\n');
```

原型为两个参数提供的默认值为 1 和 EOF，该函数的返回类型为 istream &：

```
istream & ignore(int = 1, int = EOF);
```

默认参数值 EOF 导致 ignore( )读取指定数目的字符或读取到文件尾。

该函数返回调用对象，这使得能够拼接函数调用，如下所示：

```
cin.ignore(255, '\n').ignore(255, '\n');
```

其中，第一个 ignore( )方法读取并丢弃一行，第二个调用读取并丢弃另一行，因此一共读取了两行。现在来看一看程序清单 17.13。

程序清单 17.13　get_gun.cpp

```
// get_fun.cpp -- using get() and getline()
#include <iostream>
const int Limit = 255;

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

    char input[Limit];

    cout << "Enter a string for getline() processing:\n";
    cin.getline(input, Limit, '#');
    cout << "Here is your input:\n";
    cout << input << "\nDone with phase 1\n";

    char ch;
    cin.get(ch);
    cout << "The next input character is " << ch << endl;

    if (ch != '\n')
        cin.ignore(Limit, '\n');    // discard rest of line

    cout << "Enter a string for get() processing:\n";
    cin.get(input, Limit, '#');
    cout << "Here is your input:\n";
    cout << input << "\nDone with phase 2\n";

    cin.get(ch);
    cout << "The next input character is " << ch << endl;

    return 0;
}
```

下面是程序清单 17.13 中程序的运行情况：

```
Enter a string for getline() processing:
Please pass
me a #3 melon!
Here is your input:
Please pass
me a
Done with phase 1
The next input character is 3
Enter a string for get() processing:
I still
want my #3 melon!
```

```
Here is your input:
I still
want my
Done with phase 2
The next input character is #
```

注意，getline( )函数将丢弃输入中的分界字符#，而 get( )函数不会。

4. 意外字符串输入

get(char *, int)和 getline( )的某些输入形式将影响流状态。与其他输入函数一样，这两个函数在遇到文件尾时将设置 eofbit，遇到流被破坏（如设备故障）时将设置 badbit。另外两种特殊情况是无输入以及输入到达或超过函数调用指定的最大字符数。下面来看这些情况。

对于上述两个方法，如果不能抽取字符，它们将把空值字符放置到输入字符串中，并使用 setstate( )设置 failbit。方法在什么时候无法抽取字符呢？一种可能的情况是输入方法立刻到达了文件尾。对于 get(char *, int)来说，另一种可能是输入了一个空行：

```
char temp[80];
while (cin.get(temp,80))  // terminates on empty line
      ...
```

有意思的是，空行并不会导致 getline( )设置 failbit。这是因为 getline( )仍将抽取换行符，虽然不会存储它。如果希望 getline( )在遇到空行时终止循环，则可以这样编写：

```
char temp[80];
while (cin.getline(temp,80) && temp[0] != '\0') // terminates on empty line
```

现在假设输入队列中的字符数等于或超过了输入方法指定的最大字符数。首先，来看 getline( )和下面的代码：

```
char temp[30];
while (cin.getline(temp,30))
```

getline( )方法将从输入队列中读取字符，将它们放到 temp 数组的元素中，直到（按测试顺序）到达文件尾、将要读取的字符是换行符或存储了 29 个字符为止。如果遇到文件尾，则设置 eofbit；如果将要读取的字符是换行符，则该字符将被读取并丢弃；如果读取了 29 个字符，并且下一个字符不是换行符，则设置 failbit。因此，包含 30 个或更多字符的输入行将终止输入。

现在来看 get(char *, int)方法。它首先测试字符数，然后测试是否为文件尾以及下一个字符是否是换行符。如果它读取了最大数目的字符，则不设置 failbit 标记。然而，由此可以知道终止读取是否是由于输入字符过多引起的。可以用 peek( )（参见下一节）来查看下一个输入字符。如果它是换行符，则说明 get( )已读取了整行；如果不是换行符，则说明 get( )是在到达行尾前停止的。这种技术对 getline( )不适用，因为 getline( )读取并丢弃换行符，因此查看下一个字符无法知道任何情况。然而，如果使用的是 get( )，则可以知道是否读取了整个一行。下一节将介绍这种方法的一个例子。另外，表 17.6 总结了这些行为。

表 17.6　输入行为

方　法	行　为
getline(char *, int)	如果没有读取任何字符（但换行符被视为读取了一个字符），则设置 failbit 如果读取了最大数目的字符，且行中还有其他字符，则设置 failbit
get(char *, int)	如果没有读取任何字符，则设置 failbit

17.3.4　其他 istream 方法

除前面讨论过的外，其他 istream 方法包括 read( )、peek( )、gcount( )和 putback( )。read( )函数读取指定数目的字节，并将它们存储在指定的位置中。例如，下面的语句从标准输入中读取 144 个字符，并将它

们存储在 gross 数组中：

```
char gross[144];
cin.read(gross, 144);
```

与 getline( )和 get( )不同的是，read( )不会在输入后加上空值字符，因此不能将输入转换为字符串。read( )方法不是专为键盘输入设计的，它最常与 ostream write( )函数结合使用，来完成文件输入和输出。该方法的返回类型为 istream &，因此可以像下面这样将它拼接起来：

```
char gross[144];
char score[20];
cin.read(gross, 144).read(score, 20);
```

peek( )函数返回输入中的下一个字符，但不抽取输入流中的字符。也就是说，它使得能够查看下一个字符。假设要读取输入，直到遇到换行符或句点，则可以用 peek( )查看输入流中的下一个字符，以此来判断是否继续读取：

```
char great_input[80];
char ch;
int i = 0;
while ((ch = cin.peek()) != '.' && ch != '\n')
    cin.get(great_input[i++]);
great_input [i] = '\0';
```

cin.peek( )查看下一个输入字符，并将它赋给 ch。然后，while 循环的测试条件检查 ch 是否是句点或换行符。如果是，循环将该字符读入到数组中，并更新数组索引。当循环终止时，句点和换行符将留在输入流中，并作为接下来的输入操作读取的第一个字符。然后，代码将一个空值字符放在数组的最后，使之成为一个字符串。

gcount( )方法返回最后一个非格式化抽取方法读取的字符数。这意味着字符是由 get( )、getline( )、ignore( )或 read( )方法读取的，不是由抽取运算符（>>）读取的，抽取运算符对输入进行格式化，使之与特定的数据类型匹配。例如，假设使用 cin.get（myarray，80）将一行读入 myarray 数组中，并想知道读取了多少个字符，则可以使用 strlen( )函数来计算数组中的字符数，这种方法比使用 cin.gcount( )计算从输入流中读取了多少字符的速度要快。

putback( )函数将一个字符插入到输入字符串中，被插入的字符将是下一条输入语句读取的第一个字符。putback( )方法接受一个 char 参数——要插入的字符，其返回类型为 istream &，这使得可以将该函数调用与其他 istream 方法拼接起来。使用 peek( )的效果相当于先使用 get( )读取一个字符，然后使用 putback( )将该字符放回到输入流中。然而，putback( )允许将字符放到不是刚才读取的位置。

程序清单 17.14 采用两种方式来读取并显示输入中#字符（不包括）之前的内容。第一种方法读取#字符，然后使用 putback( )将它插回到输入中。第二种方法在读取之前使用 peek( )查看下一个字符。

程序清单 17.14　peeker.cpp

```
// peeker.cpp -- some istream methods
#include <iostream>

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

//  read and echo input up to a # character
    char ch;

    while(cin.get(ch))          // terminates on EOF
```

```
    {
        if (ch != '#')
            cout << ch;
        else
        {
            cin.putback(ch);    // reinsert character
            break;
        }
    }

    if (!cin.eof())
    {
        cin.get(ch);
        cout << endl << ch << " is next input character.\n";
    }
    else
    {
        cout << "End of file reached.\n";
        std::exit(0);
    }

    while(cin.peek() != '#')    // look ahead
    {
        cin.get(ch);
        cout << ch;
    }
    if (!cin.eof())
    {
        cin.get(ch);
        cout << endl << ch << " is next input character.\n";
    }
    else
        cout << "End of file reached.\n";

    return 0;

}
```

下面是程序清单 17.14 中程序的运行情况：

```
I used a #3 pencil when I should have used a #2.
I used a
# is next input character.
3 pencil when I should have used a
# is next input character.
```

程序说明

来详细讨论程序清单 17.14 中的一些代码。第一种方法是用 while 循环来读取输入：

```
while(cin.get(ch))              // terminates on EOF
{
    if (ch != '#')
        cout << ch;
    else
    {
```

```
        cin.putback(ch);  // reinsert character
        break;
    }
}
```

达到文件尾时，表达式（cin.get（ch））将返回 false，因此从键盘模拟文件尾将终止循环。如果#字符首先出现，则程序将该字符放回到输入流中，并使用 break 语句来终止循环。

第二种方法看上去更简单：

```
while(cin.peek() != '#')      // look ahead
{
    cin.get(ch);
    cout << ch;
}
```

程序查看下一个字符。如果它不是#，则读取并显示它，然后再查看下一个字符。这一过程将一直继续下去，直到出现分界字符。

现在来看一个例子（参见程序清单 17.15），它使用 peek( )来确定是否读取了整行。如果一行中只有部分内容被加入到输入数组中，程序将删除余下的内容。

程序清单 17.15　truncate.cpp

```
// truncate.cpp -- using get() to truncate input line, if necessary
#include <iostream>
const int SLEN = 10;
inline void eatline() { while (std::cin.get() != '\n') continue; }
int main()
{
    using std::cin;
    using std::cout;
    using std::endl;

    char name[SLEN];
    char title[SLEN];
    cout << "Enter your name: ";
    cin.get(name,SLEN);
    if (cin.peek() != '\n')
        cout << "Sorry, we only have enough room for "
                << name << endl;
    eatline();
    cout << "Dear " << name << ", enter your title: \n";
    cin.get(title,SLEN);
    if (cin.peek() != '\n')
        cout << "We were forced to truncate your title.\n";
    eatline();
    cout << " Name: " << name
         << "\nTitle: " << title << endl;

    return 0;

}
```

下面是程序清单 17.15 中程序的运行情况：

```
Enter your name: Ella Fishsniffer
Sorry, we only have enough room for Ella Fish
Dear Ella Fish, enter your title:
```

```
Executive Adjunct
We were forced to truncate your title.
 Name: Ella Fish
Title: Executive
```

注意，下面的代码确定第一条输入语句是否读取了整行：

```
while (cin.get() != '\n') continue;
```

如果 get( )读取了整行，它将保留换行符，而上述代码将读取并丢弃换行符。如果 get( )只读取一部分，则上述代码将读取并丢弃该行中余下的内容。如果不删除余下的内容，则下一条输入语句将从第一个输入行中余下部分的开始位置读取。对于这个例子，这将导致程序把字符串 sniffer 读取到 title 数组中。

17.4 文件输入和输出

大多数计算机程序都使用了文件。字处理程序创建文档文件；数据库程序创建和搜索信息文件；编译器读取源代码文件并生成可执行文件。文件本身是存储在某种设备（磁带、光盘、软盘或硬盘）上的一系列字节。通常，操作系统管理文件，跟踪它们的位置、大小、创建时间等。除非在操作系统级别上编程，否则通常不必担心这些事情。需要的只是将程序与文件相连的途径、让程序读取文件内容的途径以及让程序创建和写入文件的途径。重定向（本章前面讨论过）可以提供一些文件支持，但它比显式程序中的文件 I/O 的局限性更大。另外，重定向来自操作系统，而非 C++，因此并非所有系统都有这样的功能。本书前面简要地介绍过文件 I/O，本章将更详细地探讨这个主题。

C++ I/O 类软件包处理文件输入和输出的方式与处理标准输入和输出的方式非常相似。要写入文件，需要创建一个 ofstream 对象，并使用 ostream 方法，如<<插入运算符或 write( )。要读取文件，需要创建一个 ifstream 对象，并使用 istream 方法，如>>抽取运算符或 get( )。然而，与标准输入和输出相比，文件的管理更为复杂。例如，必须将新打开的文件和流关联起来。可以以只读模式、只写模式或读写模式打开文件。写文件时，可能想创建新文件、取代旧文件或添加到旧文件中，还可能想在文件中来回移动。为帮助处理这些任务，C++在头文件 fstream（以前为 fstream.h）中定义了多个新类，其中包括用于文件输入的 ifstream 类和用于文件输出的 ofstream 类。C++还定义了一个 fstream 类，用于同步文件 I/O。这些类都是从头文件 iostream 中的类派生而来的，因此这些新类的对象可以使用前面介绍过的方法。

17.4.1 简单的文件 I/O

要让程序写入文件，必须这样做：

1. 创建一个 ofstream 对象来管理输出流；
2. 将该对象与特定的文件关联起来；
3. 以使用 cout 的方式使用该对象，唯一的区别是输出将进入文件，而不是屏幕。

要完成上述任务，首先应包含头文件 fstream。对于大多数（但不是全部）实现来说，包含该文件便自动包括 iostream 文件，因此不必显示包含 iostream。然后声明一个 ofstream 对象：

```
ofstream fout;       // create an ofstream object named fout
```

对象名可以是任意有效的 C++名称，如 fout、outFile、cgate 或 didi。

接下来，必须将这个对象与特定的文件关联起来。为此，可以使用 open( )方法。例如，假设要打开文件 jar.txt 进行输出，则可以这样做：

```
fout.open("jar.txt");  // associate fout with jar.txt
```

可以使用另一个构造函数将这两步（创建对象和关联到文件）合并成一条语句：

```
ofstream fout("jar.txt");  // create fout object, associate it with jar.txt
```

然后，以使用 cout 的方式使用 fout（或选择的其他名称）。例如，要将 Dull Data 放到文件中，可以这

样做：

```
fout << "Dull Data";
```

由于 ostream 是 ofstream 类的基类，因此可以使用所有的 ostream 方法，包括各种插入运算符定义、格式化方法和控制符。ofstream 类使用被缓冲的输出，因此程序在创建像 fout 这样的 ofstream 对象时，将为输出缓冲区分配空间。如果创建了两个 ofstream 对象，程序将创建两个缓冲区，每个对象各一个。像 fout 这样的 ofstream 对象从程序那里逐字节地收集输出，当缓冲区填满后，它便将缓冲区内容一同传输给目标文件。由于磁盘驱动器被设计称以大块的方式传输数据，而不是逐字节地传输，因此通过缓冲可以大大提高从程序到文件传输数据的速度。

以这种方式打开文件来进行输出时，如果没有这样的文件，将创建一个新文件；如果有这样的文件，则打开文件将清空文件，输出将进入到一个空文件中。本章后面将介绍如何打开已有的文件，并保留其内容。

警告：以默认模式打开文件进行输出将自动把文件的长度截短为零，这相当于删除已有的内容。

读取文件的要求与写入文件相似：

- 创建一个 ifstream 对象来管理输入流；
- 将该对象与特定的文件关联起来；
- 以使用 cin 的方式使用该对象。

上述读文件的步骤类似于写文件。首先，当然要包含头文件 fstream。然后声明一个 ifstream 对象，将它与文件名关联起来。可以使用一两条语句来完成这项工作：

```
// two statements
ifstream fin;              // create ifstream object called fin
fin.open("jellyjar.txt");  // open jellyjar.txt for reading
// one statement
ifstream fis("jamjar.txt"); // create fis and associate with jamjar.txt
```

现在，可以像使用 cin 那样使用 fin 或 fis。例如，可以这样做：

```
char ch;
fin >> ch;                 // read a character from the jellyjar.txt file
char buf[80];
fin >> buf;                // read a word from the file
fin.getline(buf, 80);      // read a line from the file
string line;
getline(fin, line);        // read from a file to a string object
```

输入和输出一样，也是被缓冲的，因此创建 ifstream 对象与 fin 一样，将创建一个由 fin 对象管理的输入缓冲区。与输出一样，通过缓冲，传输数据的速度比逐字节传输要快得多。

当输入和输出流对象过期（如程序终止）时，到文件的连接将自动关闭。另外，也可以使用 close( ) 方法来显式地关闭到文件的连接：

```
fout.close();     // close output connection to file
fin.close();      // close input connection to file
```

关闭这样的连接并不会删除流，而只是断开流到文件的连接。然而，流管理装置仍被保留。例如，fin 对象与它管理的输入缓冲区仍然存在。您稍后将知道，可以将流重新连接到同一个文件或另一个文件。

我们来看一个简短的例子。程序清单 17.16 的程序要求输入文件名，然后创建一个名称为输入名的文件，将一些信息写入到该文件中，然后关闭该文件。关闭文件将刷新缓冲区，从而确保文件被更新。然后，程序打开该文件，读取并显示其内容。注意，该程序以使用 cin 和 cout 的方式使用 fin 和 fout。另外，该程序将文件名读取到一个 string 对象中，然后使用方法 c_str( )来给 ofstream 和 ifstream 的构造函数提供一个 C-风格字符串参数。

程序清单 17.16 fileio.cpp

```
// fileio.cpp -- saving to a file
#include <iostream> // not needed for many systems
#include <fstream>
#include <string>

int main()
{
    using namespace std;
    string filename;

    cout << "Enter name for new file: ";
    cin >> filename;

// create output stream object for new file and call it fout
    ofstream fout(filename.c_str());

    fout << "For your eyes only!\n";        // write to file
    cout << "Enter your secret number: ";   // write to screen
    float secret;
    cin >> secret;
    fout << "Your secret number is " << secret << endl;
    fout.close();           // close file

// create input stream object for new file and call it fin
    ifstream fin(filename.c_str());
    cout << "Here are the contents of " << filename << ":\n";
    char ch;
    while (fin.get(ch))     // read character from file and
        cout << ch;         // write it to screen
    cout << "Done\n";
    fin.close();

    return 0;
}
```

下面是程序清单 17.16 中程序的运行情况：

```
Enter name for new file: pythag
Enter your secret number: 3.14159
Here are the contents of pythag:
For your eyes only!
Your secret number is 3.14159
Done
```

如果查看该程序所在的目录，将看到一个名为 pythag 的文件，使用文本编辑器打开该文件，其内容将与程序输出相同。

17.4.2 流状态检查和 is_open( )

C++文件流类从 ios_base 类那里继承了一个流状态成员。正如前面指出的，该成员存储了指出流状态的信息：一切顺利、已到达文件尾、I/O 操作失败等。如果一切顺利，则流状态为零（没有消息就是好消息）。其他状态都是通过将特定位设置为 1 来记录的。文件流类还继承了 ios_base 类中报告流状态的方法，表 17.4 对这些方法进行了总结。可以通过检查流状态来判断最后一个流操作是否成功。对于文件流，这包

括检查试图打开文件时是否成功。例如，试图打开一个不存在的文件进行输入时，将设置 failbit 位，因此可以这样进行检查：

```
fin.open(argv[file]);
if (fin.fail())  // open attempt failed
{
    ...
}
```

由于 ifstream 对象和 istream 对象一样，被放在需要 bool 类型的地方时，将被转换为 bool 值，因此您也可以这样做：

```
fin.open(argv[file]);
if (!fin)  // open attempt failed
{
    ...
}
```

然而，较新的 C++实现提供了一种更好的检查文件是否被打开的方法——is_open( )方法：

```
if (!fin.is_open())  // open attempt failed
{
    ...
}
```

这种方式之所以更好，是因为它能够检测出其他方式不能检测出的微妙问题，接下来的“警告”将讨论这一点。

警告：以前，检查文件是否成功打开的常见方式如下：

```
if(fin.fail()) ...  // failed to open
if(!fin.good()) ... // failed to open
if (!fin) ...       // failed to open
```

fin 对象被用于测试条件中时，如果 fin.good( )为 false，将被转换为 false；否则将被转换为 true。因此上面三种方式等价。然而，这些测试无法检测到这样一种情形：试图以不合适的文件模式（参见本章后面的“文件模式”一节）打开文件时失败。方法 is_open( )能够检测到这种错误以及 good( )能够检测到的错误。然而，老式 C++实现没有 is_open( )。

17.4.3　打开多个文件

程序可能需要打开多个文件。打开多个文件的策略取决于它们将被如何使用。如果需要同时打开两个文件，则必须为每个文件创建一个流。例如，将两个排序后的文件拼接成第三个文件的程序，需要为两个输入文件创建两个 ifstream 对象，并为输出文件创建一个 ofstream 对象。可以同时打开的文件数取决于操作系统。

然而，可能要依次处理一组文件。例如，可能要计算某个名称在 10 个文件中出现的次数。在这种情况下，可以打开一个流，并将它依次关联到各个文件。这在节省计算机资源方面，比为每个文件打开一个流的效率高。使用这种方法，首先需要声明一个 ifstream 对象（不对它进行初始化），然后使用 open( )方法将这个流与文件关联起来。例如，下面是依次读取两个文件的代码：

```
ifstream fin;           // create stream using default constructor
fin.open("fat.txt");    // associate stream with fat.txt file
...                     // do stuff
fin.close();            // terminate association with fat.txt
fin.clear();            // reset fin (may not be needed)
fin.open("rat.txt");    // associate stream with rat.txt file
...
fin.close();
```

稍后将介绍一个例子，但先来看这样一种将一系列文件输入给程序的技术，即让程序能够使用循环来处理文件。

17.4.4 命令行处理技术

文件处理程序通常使用命令行参数来指定文件。命令行参数是用户在输入命令时，在命令行中输入的参数。例如，要在 UNIX 或 Linux 系统中计算文件包含的字数，可以在命令行提示符下输入下面的命令：

```
wc report1 report2 report3
```

其中，wc 是程序名，report1、report2 和 report3 是作为命令行参数传递给程序的文件名。

C++有一种让在命令行环境中运行的程序能够访问命令行参数的机制，方法是使用下面的 main( ) 函数：

```
int main(int argc, char *argv[])
```

argc 为命令行中的参数个数，其中包括命令名本身。argv 变量为一个指针，它指向一个指向 char 的指针。这过于抽象，但可以将 argv 看作一个指针数组，其中的指针指向命令行参数，argv[0]是一个指针，指向存储第一个命令行参数的字符串的第一个字符，依此类推。也就是说，argv[0]是命令行中的第一个字符串，依此类推。例如，假设有下面的命令行：

```
wc report1 report2 report3
```

则 argc 为 4，argv[0]为 wc，argv[1]为 report1，依此类推。下面的循环将把每个命令行参数分别打印在单独一行上：

```
for (int i = 1; i < argc; i++)
        cout << argv[i] << endl;
```

以 i=1 开头将只打印命令行参数；以 i=0 开头将同时打印命令名。

当然，命令行参数与命令行操作系统（如 Windows 命令提示符模式、UNIX 和 Linux）紧密相关。其他程序也可能允许使用命令行参数。

- 很多 Windows IDE（集成开发环境）都有一个提供命令行参数的选项。通常，必须选择一系列菜单，才能打开一个可以输入命令行参数的对话框。具体的步骤随厂商和升级版本而异，因此请查看文档。
- 很多 Windows IDE 都可以生成可执行文件，这些文件能够在 Windows 命令提示符模式下运行。

程序清单 17.17 结合使用命令行技术和文件流技术，来计算命令行上列出的文件包含的字符数。

程序清单 17.17 count.cpp

```
// count.cpp -- counting characters in a list of files
#include <iostream>
#include <fstream>
#include <cstdlib>   // for exit()
int main(int argc, char * argv[])
{
    using namespace std;
    if (argc == 1)          // quit if no arguments
    {
        cerr << "Usage: " << argv[0] << " filename[s]\n";
        exit(EXIT_FAILURE);
    }

    ifstream fin;               // open stream
    long count;
    long total = 0;
```

```
    char ch;

    for (int file = 1; file < argc; file++)
    {
        fin.open(argv[file]);  // connect stream to argv[file]
        if (!fin.is_open())
        {
            cerr << "Could not open " << argv[file] << endl;
            fin.clear();
            continue;
        }
        count = 0;
        while (fin.get(ch))
            count++;
        cout << count << " characters in " << argv[file] << endl;
        total += count;
        fin.clear();            // needed for some implementations
        fin.close();            // disconnect file
    }
    cout << total << " characters in all files\n";

    return 0;
}
```

注意：有些 C++实现要求在该程序末尾使用 fin.clear( )，有些则不要求，这取决于将文件与 ifstream 对象关联起来时，是否自动重置流状态。使用 fin.clear( )是无害的，即使在不必使用它的时候使用。

例如，在 Linux 系统中，可以将程序清单 17.17 编译为一个名为 a.out 的可执行文件。该程序的运行情况如下：

```
$ a.out
Usage: a.out filename[s]
$ a.out paris rome
3580 characters in paris
4886 characters in rome
8466 characters in all files
$
```

注意，该程序使用 cerr 表示错误消息。另外，消息使用 argv[0]，而不是 a.out：

```
cerr << "Usage: " << argv[0] << " filename[s]\n";
```

如果修改了可执行文件的名称，则程序将自动使用新的名称。

该程序使用 is_open( )方法来确定能够打开指定的文件，下面更深入地探讨这一主题。

17.4.5　文件模式

文件模式描述的是文件将被如何使用：读、写、追加等。将流与文件关联时（无论是使用文件名初始化文件流对象，还是使用 open( )方法），都可以提供指定文件模式的第二个参数：

```
ifstream fin("banjo", mode1);  // constructor with mode argument
ofstream fout();
fout.open("harp", mode2);      // open() with mode arguments
```

ios_base 类定义了一个 openmode 类型，用于表示模式；与 fmtflags 和 iostate 类型一样，它也是一种 bitmask 类型（以前，其类型为 int）。可以选择 ios_base 类中定义的多个常量来指定模式，表 17.7 列出了这些常量及其含义。C++文件 I/O 作了一些改动，以便与 ANSI C 文件 I/O 兼容。

表 17.7 文件模式常量

常　量	含　义
ios_base::in	打开文件，以便读取
ios_base::out	打开文件，以便写入
ios_base::ate	打开文件，并移到文件尾
ios_base::app	追加到文件尾
ios_base::trunc	如果文件存在，则截短文件
ios_base::binary	二进制文件

如果 ifstream 和 ofstream 构造函数以及 open( )方法都接受两个参数，为什么前面的例子只使用一个参数就可以调用它们呢？您可能猜到了，这些类成员函数的原型为第二个参数（文件模式参数）提供了默认值。例如，ifstream open( )方法和构造函数用 ios_base::in（打开文件以读取）作为模式参数的默认值，而 ofstream open( )方法和构造函数用 ios_base::out | ios_base::trunc（打开文件，以读取并截短文件）作为默认值。位运算符 OR（1）用于将两个位值合并成一个可用于设置两个位的值。fstream 类不提供默认的模式值，因此在创建这种类的对象时，必须显式地提供模式。

注意，ios_base::trunc 标记意味着打开已有的文件，以接收程序输出时将被截短；也就是说，其以前的内容将被删除。虽然这种行为极大地降低了耗尽磁盘空间的危险，但您也许能够想象到这样的情形，即不希望打开文件时将其内容删除。当然，C++提供了其他的选择。例如，如果要保留文件内容，并在文件尾添加（追加）新信息，则可以使用 ios_base::app 模式：

```
ofstream fout("bagels", ios_base::out | ios_base::app);
```

上述代码也使用|运算符来合并模式，因此 ios_base::out | ios_base::app 意味着启用模式 out 和 app（参见图 17.6）。

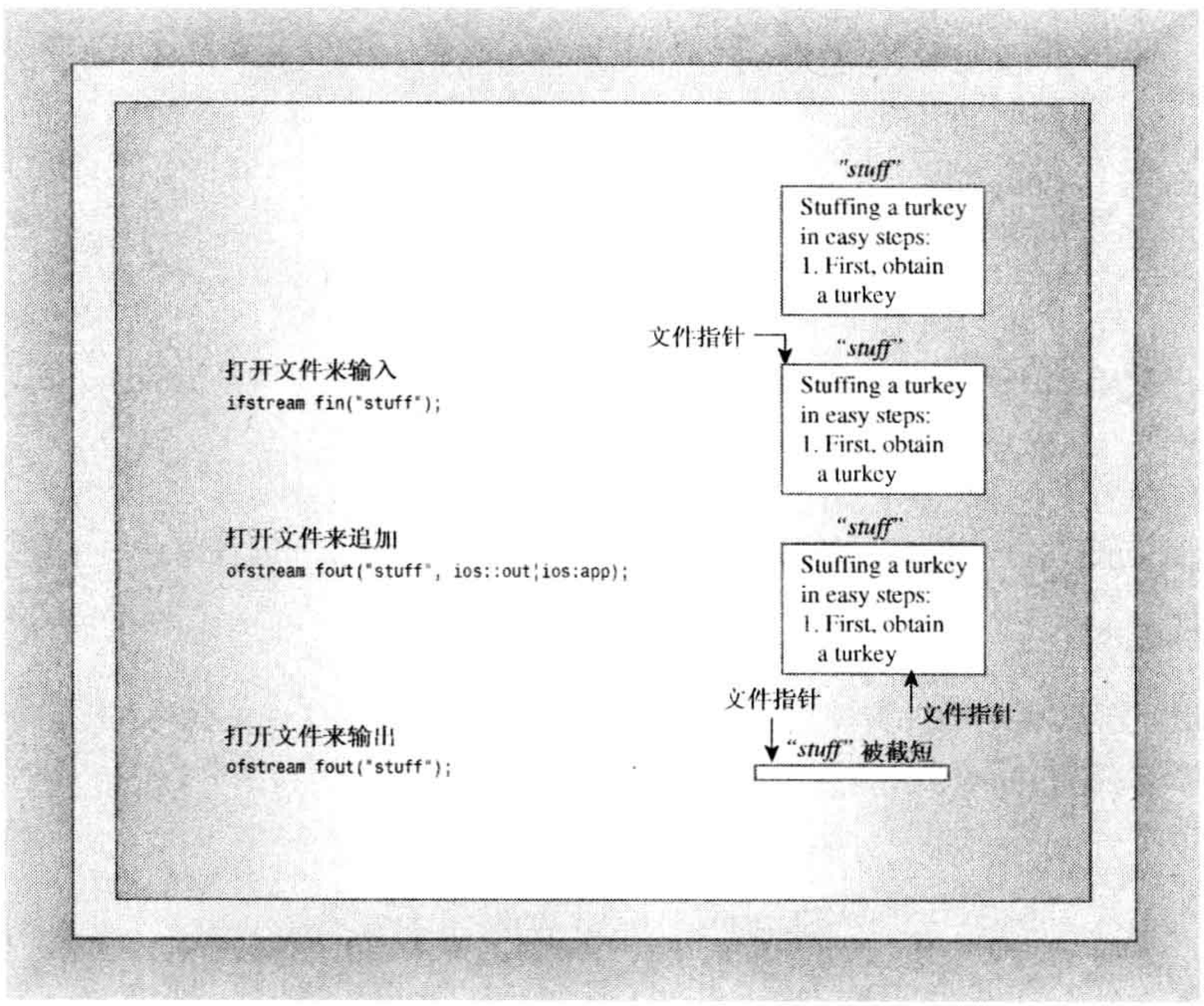

图 17.6　一些文件打开模式

老式 C++实现之间可能有一些差异。例如，有些实现允许省略前一例子中的 ios_base::out，有些则不

允许。如果不使用默认模式，则最安全的方法是显式地提供所有的模式元素。有些编译器不支持表 17.6 中的所有选项，有些则提供了表中没有列出的其他选项。这些差异导致的后果之一是，可能必须对后面的例子作一些修改，使之能够在所用的系统中运行。好在 C++标准提供了更高的统一性。

标准 C++根据 ANSI C 标准 I/O 定义了部分文件 I/O。实现像下面这样的 C++语句时：

```
ifstream fin(filename, c++mode);
```

就像它使用了 C 的 fopen( )函数一样：

```
fopen(filename, cmode);
```

其中，c++mode 是一个 openmode 值，如 ios_base::in；而 cmode 是相应的 C 模式字符串，如“r”。表 17.8 列出了 C++模式和 C 模式的对应关系。注意，ios_base::out 本身将导致文件被截短，但与 ios_base::in 一起使用时，不会导致文件被截短。没有列出的组合，如 ios_base::in [vn] ios_base::trunc，将禁止文件被打开。is_open( )方法用于检测这种故障。

表 17.8　C++和 C 的文件打开模式

C++模式	C 模式	含　义
ios_base :: in	"r"	打开以读取
ios_base :: out	"w"	等价于 ios_base :: out | ios_base :: trunc
ios_base :: out | ios_base :: trunc	"w"	打开以写入，如果已经存在，则截短文件
ios_base :: out | ios_base :: app	"a"	打开以写入，只追加
ios_base :: out | ios_base :: out	"r+"	打开以读写，在文件允许的位置写入
ios_base :: out | ios_base :: out | ios_base::trunc	"w+"	打开以读写，如果已经存在，则首先截短文件
c++mode | ios_base :: binary	"cmodeb"	以 C++mode（或相应的 cmode）和二进制模式打开；例如，ios_base :: in | ios_base :: binary 成为“rb”
c++mode | ios_base :: ate	"cmode"	以指定的模式打开，并移到文件尾。C 使用一个独立的函数调用，而不是模式编码。例如，ios_base :: in | ios_base :: ate 被转换为“r”模式和 C 函数调用 fseek(file, 0, SEEK_END)

注意，ios_base::ate 和 ios_base::app 都将文件指针指向打开的文件尾。二者的区别在于，ios_base::app 模式只允许将数据添加到文件尾，而 ios_base::ate 模式将指针放到文件尾。

显然，各种模式的组合很多，我们将介绍几种有代表性的组合。

1. 追加文件

来看一个在文件尾追加数据的程序。该程序维护一个存储来客清单的文件。该程序首先显示文件当前的内容（如果有话）。在尝试打开文件后，它使用 is_open( )方法来检查该文件是否存在。接下来，程序以 ios_base::app 模式打开文件，进行输出。然后，它请求用户从键盘输入，并将其添加到文件中。最后，程序显示修订后的文件内容。程序清单 17.18 演示了如何实现这些目标。请注意程序是如何使用 is_open( )方法来检测文件是否被成功打开的。

注意：在早期，文件 I/O 可能是 C++最不标准的部分，很多老式编译器都不遵守当前的标准。例如，有些编译器使用诸如 nocreate 等模式，而这些模式不是当前标准的组成部分。另外，只有一部分编译器要求在第二次打开同一个文件进行读取之前调用 fin.clear( )。

程序清单 17.18　append.cpp

```
// append.cpp -- appending information to a file
#include <iostream>
#include <fstream>
#include <string>
```

```
#include <cstdlib>      // (for exit()

const char * file = "guests.txt";
int main()
{
    using namespace std;
    char ch;
// show initial contents
    ifstream fin;
    fin.open(file);

    if (fin.is_open())
    {
        cout << "Here are the current contents of the "
             << file << " file:\n";
        while (fin.get(ch))
            cout << ch;
        fin.close();
    }

// add new names
    ofstream fout(file, ios::out | ios::app);
    if (!fout.is_open())
    {
        cerr << "Can't open " << file << " file for output.\n";
        exit(EXIT_FAILURE);
    }

    cout << "Enter guest names (enter a blank line to quit):\n";
    string name;
    while (getline(cin,name) && name.size() > 0)
    {
          fout << name << endl;
    }
    fout.close();

// show revised file
    fin.clear();    // not necessary for some compilers
    fin.open(file);
    if (fin.is_open())
    {
        cout << "Here are the new contents of the "
             << file << " file:\n";
        while (fin.get(ch))
            cout << ch;
        fin.close();
    }
    cout << "Done.\n";
    return 0;

}
```

下面是第一次运行程序清单 17.18 中程序的情况：

```
Enter guest names (enter a blank line to quit):
Genghis Kant
```

```
Hank Attila
Charles Bigg

Here are the new contents of the guests.txt file:
Genghis Kant
Hank Attila
Charles Bigg
Done.
```

此时，guests.txt 文件还没有创建，因此程序不能预览该文件。

但第二次运行该程序时，guests.txt 文件已经存在，因此程序将预览该文件。另外，新数据被追加到旧文件的后面，而不是取代它们。

```
Here are the current contents of the guests.txt file:
Genghis Kant
Hank Attila
Charles Bigg
Enter guest names (enter a blank line to quit):
Greta Greppo
LaDonna Mobile
Fannie Mae

Here are the new contents of the guests.txt file:
Ghengis Kant
Hank Attila
Charles Bigg
Greta Greppo
LaDonna Mobile
Fannie Mae
Done.
```

可以用任何文本编辑器来读取 guest.txt 的内容，包括用来编写源代码的编辑器。

2. 二进制文件

将数据存储在文件中时，可以将其存储为文本格式或二进制格式。文本格式指的是将所有内容（甚至数字）都存储为文本。例如，以文本格式存储值-2.324216e+07 时，将存储该数字包含的 13 个字符。这需要将浮点数的计算机内部表示转换为字符格式，这正是<<插入运算符完成的工作。另一方面，二进制格式指的是存储值的计算机内部表示。也就是说，计算机不是存储字符，而是存储这个值的 64 位 double 表示。对于字符来说，二进制表示与文本表示是一样的，即字符的 ASCII 码的二进制表示。对于数字来说，二进制表示与文本表示有很大的差别（参见图 17.7）。

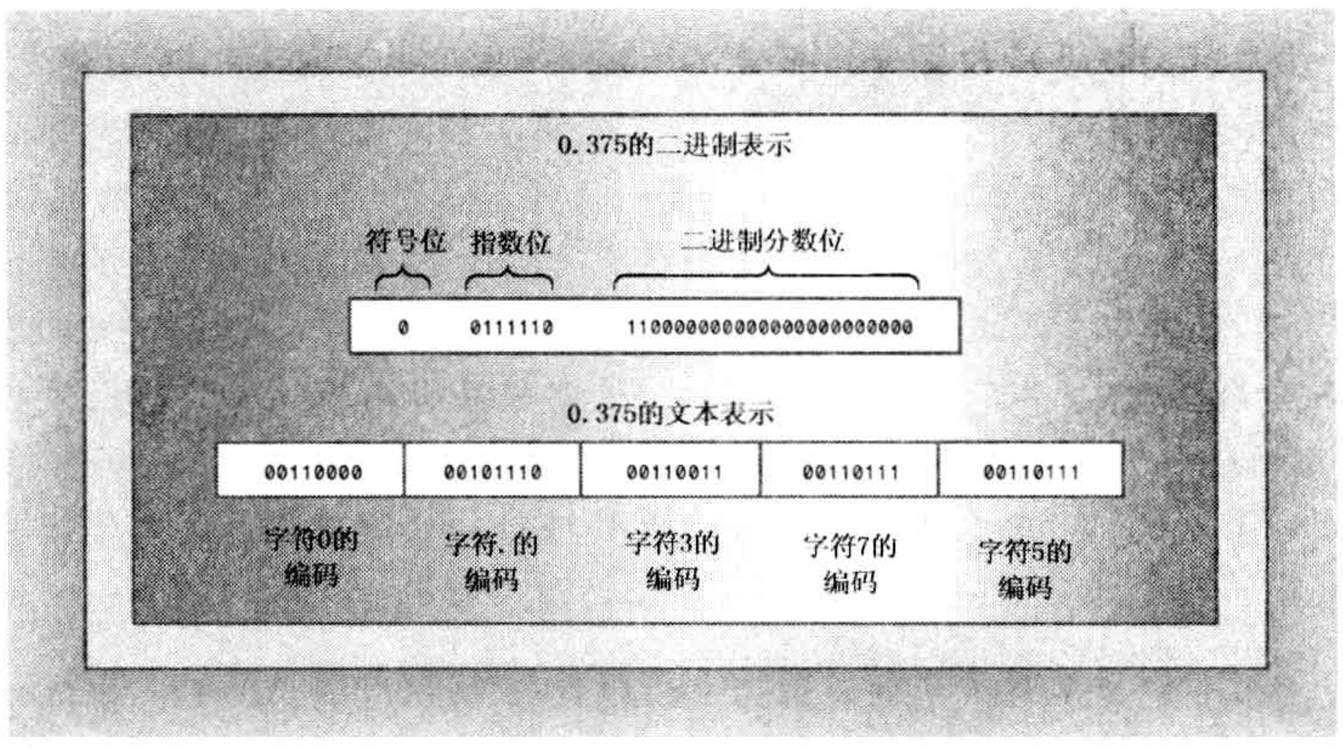

图 17.7　浮点数的二进值表示和文本表示

每种格式都有自己的优点。文本格式便于读取，可以使用编辑器或字处理器来读取和编辑文本文件，可以很方便地将文本文件从一个计算机系统传输到另一个计算机系统。二进制格式对于数字来说比较精确，因为它存储的是值的内部表示，因此不会有转换误差或舍入误差。以二进制格式保存数据的速度更快，因为不需要转换，并可以大块地存储数据。二进制格式通常占用的空间较小，这取决于数据的特征。然而，如果另一个系统使用另一种内部表示，则可能无法将数据传输给该系统。同一系统上不同的编译器也可能使用不同的内部结构布局表示。在这种情况下，则必须编写一个将一种数据转换成另一种的程序。

来看一个更具体的例子。考虑下面的结构定义和声明：

```
const int LIM = 20;
struct planet
{
    char name[LIM];        // name of planet
    double population;    // its population
    double g;             // its acceleration of gravity
};
planet pl;
```

要将结构 pl 的内容以文本格式保存，可以这样做：

```
ofstream fout("planets.dat", ios_base:: out | ios_base::app);
fout << pl.name << " " << pl.population << " " << pl.g << "\n";
```

必须使用成员运算符显式地提供每个结构成员，还必须将相邻的数据分隔开，以便区分。如果结构有 30 个成员，则这项工作将很乏味。

要用二进制格式存储相同的信息，可以这样做：

```
ofstream fout("planets.dat",
              ios_base:: out | ios_base::app | ios_base::binary);
fout.write( (char *) &pl, sizeof pl);
```

上述代码使用计算机的内部数据表示，将整个结构作为一个整体保存。不能将该文件作为文本读取，但与文本相比，信息的保存更为紧凑、精确。它确实更便于键入代码。这种方法做了两个修改：

- 使用二进制文件模式；
- 使用成员函数 write( )。

下面更详细的介绍这两项修改。

有些系统（如 Windows）支持两种文件格式：文本格式和二进制格式。如果要用二进制格式保存数据，应使用二进制文件格式。在 C++中，可以将文件模式设置为 ios_base::binary 常量来完成。要知道为什么在 Windows 系统上需要完成这样的任务，请参见后面的旁注“二进制文件和文本文件”。

二进制文件和文本文件

使用二进制文件模式时，程序将数据从内存传输给文件（反之亦然）时，将不会发生任何隐藏的转换，而默认的文本模式并非如此。例如，对于 Windows 文本文件，它们使用两个字符的组合（回车和换行）表示换行符；Macintosh 文本文件使用回车来表示换行符；而 UNIX 和 Linux 文件使用换行（linefeed）来表示换行符。C++是从 UNIX 系统上发展而来的，因此也使用换行（linefeed）来表示换行符。为增加可移植性，Windows C++程序在写文本模式文件时，自动将 C++换行符转换为回车和换行；Macintosh C++程序在写文件时，将换行符转换为回车。在读取文本文件时，这些程序将本地换行符转换为 C++格式。对于二进制数据，文本格式会引起问题，因此 double 值中间的字节可能与换行符的 ASCII 码有相同的位模式。另外，在文件尾的检测方式也有区别。因此以二进制格式保存数据时，应使用二进制文件模式（UNIX 系统只有一种文件模式，因此对于它来说，二进制模式和文本模式是一样的）。

要以二进制格式（而不是文本格式）存储数据，可以使用 write( )成员函数。前面说过，这种方法将内存中指定数目的字节复制到文件中。本章前面用它复制过文本，但它只逐字节地复制数据，而不进行任何

转换。例如，如果将一个 long 变量的地址传递给它，并命令它复制 4 个字节，它将复制 long 值中的 4 个字节，而不会将它转换为文本。唯一不方便的地方是，必须将地址强制转换为指向 char 的指针。也可以用同样的方式来复制整个 planet 结构。要获得字节数，可以使用 sizeof 运算符：

```
fout.write( (char *) &pl, sizeof pl);
```

这条语句导致程序前往 pl 结构的地址，并将开始的 36 个字节（sizeof pl 表达式的值）复制到与 fout 相关联的文件中。

要使用文件恢复信息，请通过一个 ifstream 对象使用相应的 read( )方法：

```
ifstream fin("planets.dat", ios_base::in | ios_base::binary);
fin.read((char *) &pl, sizeof pl);
```

这将从文件中复制 sizeof pl 个字节到 pl 结构中。同样的方法也适用于不使用虚函数的类。在这种情况下，只有数据成员被保存，而方法不会被保存。如果类有虚方法，则也将复制隐藏指针（该指针指向虚函数的指针表）。由于下一次运行程序时，虚函数表可能在不同的位置，因此将文件中的旧指针信息复制到对象中，将可能造成混乱（请参见“编程练习 6”中的注意）。

提示：read( )和 write( )成员函数的功能是相反的。请用 read( )来恢复用 write( )写入的数据。

程序清单 17.19 使用这些方法来创建和读取二进制文件。从形式上看，该程序与程序清单 17.18 相似，但它使用的是 write( )和 read( )，而不是插入运算符和 get( )方法。另外，它还使用控制符来格式化屏幕输出。

注意：虽然二进制文件概念是 ANSI C 的组成部分，但一些 C 和 C++实现并没有提供对二进制文件模式的支持。原因在于：有些系统只有一种文件类型，因此可以将二进制操作（如 read( )和 write( )）用于标准文件格式。因此，如果实现认为 ios_base::binary 是非法常量，只要删除它即可。如果实现不支持 fixed 和 right 控制符，则可以使用 cout.setf（ios_base::fixed、ios_base::floatfield）和 cout.setf（ios_base::right、ios_base::adjustfield）。另外，也可能必须用 ios 替换 ios_base。其他编译器（特别是老式编译器）可能还有其他特征。

程序清单 17.19　binary.cpp

```
// binary.cpp -- binary file I/O
#include <iostream> // not required by most systems
#include <fstream>
#include <iomanip>
#include <cstdlib>  // for exit()

inline void eatline() { while (std::cin.get() != '\n') continue; }
struct planet
{
    char name[20];       // name of planet
    double population;   // its population
    double g;            // its acceleration of gravity
};

const char * file = "planets.dat";

int main()
{
    using namespace std;
    planet pl;
    cout << fixed << right;

// show initial contents
```

```
    ifstream fin;
    fin.open(file, ios_base::in |ios_base::binary);  // binary file
    //NOTE: some systems don't accept the ios_base::binary mode
    if (fin.is_open())
    {
    cout << "Here are the current contents of the "
        << file << " file:\n";
    while (fin.read((char *) &pl, sizeof pl))
    {
        cout << setw(20) << pl.name << ": "
                 << setprecision(0) << setw(12) << pl.population
                 << setprecision(2) << setw(6) << pl.g << endl;
    }
    fin.close();
    }

// add new data
    ofstream fout(file,
                ios_base::out | ios_base::app | ios_base::binary);
    //NOTE: some systems don't accept the ios::binary mode
    if (!fout.is_open())
    {
        cerr << "Can't open " << file << " file for output:\n";
        exit(EXIT_FAILURE);
    }

    cout << "Enter planet name (enter a blank line to quit):\n";
    cin.get(pl.name, 20);
    while (pl.name[0] != '\0')
    {
        eatline();
        cout << "Enter planetary population: ";
        cin >> pl.population;
        cout << "Enter planet's acceleration of gravity: ";
        cin >> pl.g;
        eatline();
        fout.write((char *) &pl, sizeof pl);
        cout << "Enter planet name (enter a blank line "
                "to quit):\n";
        cin.get(pl.name, 20);
    }
    fout.close();

// show revised file
    fin.clear();    // not required for some implementations, but won't hurt
    fin.open(file, ios_base::in | ios_base::binary);
    if (fin.is_open())
    {
        cout << "Here are the new contents of the "
             << file << " file:\n";
        while (fin.read((char *) &pl, sizeof pl))
        {
            cout << setw(20) << pl.name << ": "
                 << setprecision(0) << setw(12) << pl.population
                 << setprecision(2) << setw(6) << pl.g << endl;
```

```
        }
        fin.close();
    }
    cout << "Done.\n";
    return 0;
}
```

下面是首次运行程序清单 17.19 中程序时的情况：

```
Enter planet name (enter a blank line to quit):
Earth
Enter planetary population: 6928198253
Enter planet's acceleration of gravity: 9.81
Enter planet name (enter a blank line to quit):

Here are the new contents of the planets.dat file:
               Earth:   6928198253  9.81
Done.
```

下面是再次运行该程序时的情况：

```
Here are the current contents of the planets.dat file:
               Earth:   6928198253  9.81
Enter planet name (enter a blank line to quit):
Jenny's World
Enter planetary population: 32155648
Enter planet's acceleration of gravity: 8.93
Enter planet name (enter a blank line to quit):

Here are the new contents of the planets.dat file:
               Earth:   6928198253  9.81
       Jenny's World:     32155648  8.93
Done.
```

看到该程序的主要特征后，下面再次讨论前面提到的几点。程序在读取行星的 g 值后，将使用下面的代码（以内嵌 eatline( )函数的形式）：

```
while (std::cin.get() != '\n') continue;
```

这将读取并丢弃输入中换行符之前的内容。考虑循环中的下一条输入语句：

```
cin.get(pl.name, 20);
```

如果保留换行符，该语句将换行符作为空行读取，然后终止循环。

您可能会问，如果该程序是否可以使用 string 对象而不是字符数组来表示 planet 结构的 name 成员？答案是否定的，至少在不对设计做重大修改的情况下是否定的。问题在于，string 对象本身实际上并没有包含字符串，而是包含一个指向其中存储了字符串的内存单元的指针。因此，将结构复制到文件中时，复制的将不是字符串数据，而是字符串的存储地址。当您再次运行该程序时，该地址将毫无意义。

17.4.6　随机存取

在最后一个文件示例中，将探讨随机存取。随机存取指的是直接移动（不是依次移动）到文件的任何位置。随机存取常被用于数据库文件，程序维护一个独立的索引文件，该文件指出数据在主数据文件中的位置。这样，程序便可以直接跳到这个位置，读取（还可能修改）其中的数据。如果文件由长度相同的记录组成，这种方法实现起来最简单。每条记录表示一组相关的数据。例如，在程序清单 17.19 的示例中，每条文件记录将表示关于特定行星的全部数据。很自然，文件记录对应于程序结构或类。

我们将以程序清单 17.19 中的二进制文件程序为基础，充分利用 planet 结构为文件了记录模式，来创

建这个例子。为使编程更具创造性，该示例将以读写模式打开文件，以便能够读取和修改记录。为此，可以创建一个 fstream 对象。fstream 类是从 iostream 类派生而来的，而后者基于 istream 和 ostream 两个类，因此它继承了它们的方法。它还继承了两个缓冲区，一个用于输入，一个用于输出，并能同步化这两个缓冲区的处理。也就是说，当程序读写文件时，它将协调地移动输入缓冲区中的输入指针和输出缓冲区中的输出指针。

该示例将完成以下工作：

1. 显示 planets.dat 文件当前的内容；
2. 询问要修改哪条记录；
3. 修改该记录；
4. 显示修改后的文件。

更复杂的程序将使用菜单和循环，使得能在操作列表中不断地进行选择。但这里的版本只能执行每种操作一次。这种简化让您能够检验读写文件的多个方面，而不陷入程序设计事务之中。

警告：这个程序假设 planets.dat 文件已经存在，该文件是由程序清单 17.19 中的 binary.cpp 程序创建的。

要回答的第一个问题是：应使用哪种文件模式。为读取文件，需要使用 ios_base::in 模式。为执行二进制 I/O，需要使用 ios_base::binary 模式（在某些非标准系统上，可以省略这种模式，事实上，可能必须省略这种模式）。为写入文件，需要 ios_base::out 或 ios_base::app 模式。然而，追加模式只允许程序将数据添加到文件尾，文件的其他部分是只读的；也就是说，可以读取原始数据，但不能修改它；要修改数据，必须使用 ios_base::out。表 17.8 表明，同时使用 in 模式和 out 模式将得到读/写模式，因此只需添加二进制元素即可。如前所述，要使用|运算符来组合模式。因此，需要使用下面的语句：

```
finout.open(file,ios_base::in | ios_base::out | ios_base::binary);
```

接下来，需要一种在文件中移动的方式。fstream 类为此继承了两个方法：seekg( )和 seekp( )，前者将输入指针移到指定的文件位置，后者将输出指针移到指定的文件位置（实际上，由于 fstream 类使用缓冲区来存储中间数据，因此指针指向的是缓冲区中的位置，而不是实际的文件）。也可以将 seekg( )用于 ifstream 对象，将 seekp( )用于 oftream 对象。下面是 seekg( )的原型：

```
basic_istream<charT,traits>& seekg(off_type, ios_base::seekdir);
basic_istream<charT,traits>& seekg(pos_type);
```

正如您看到的，它们都是模板。本章将使用 char 类型的模板具体化。对于 char 具体化，上面两个原型等同于下面的代码：

```
istream & seekg(streamoff, ios_base::seekdir);
istream & seekg(streampos);
```

第一个原型定位到离第二个参数指定的文件位置特定距离（单位为字节）的位置；第二个原型定位到离文件开头特定距离（单位为字节）的位置。

类型升级

在 C++早期，seekg( )方法比较简单。Streamoff 和 streampos 类型是一些标准整型（如 long）的 typedef。但为创建可移植标准，必须处理这样的现实情况：对于有些文件系统，整数参数无法提供足够的信息，因此 streamoff 和 streampos 允许是结构或类类型，条件是它们允许一些基本的操作，如使用整数值作为初始值等。随后，老版本的 istream 类被 basic_istream 模板取代，streampos 和 streamoff 被 basic_istream 模板取代。然而，streampos 和 streamoff 继续存在，作为 pos_type 和 off_type 的 char 的具体化。同样，如果将 seekg( )用于 wistream 对象，可以使用 wstreampos 和 wstreamoff 类型。

来看 seekg( )的第一个原型的参数。streamoff 值被用来度量相对于文件特定位置的偏移量（单位为字节）。streamoff 参数表示相对于三个位置之一的偏移量为特定值（以字节为单位）的文件位置（类型可定义为整型或类）。seek_dir 参数是 ios_base 类中定义的另一种整型，有 3 个可能的值。常量 ios_base::beg 指

相对于文件开始处的偏移量。常量 ios_base::cur 指相对于当前位置的偏移量；常量 ios_base::end 指相对于文件尾的偏移量。下面是一些调用示例，这里假设 fin 是一个 ifstream 对象：

```
fin.seekg(30, ios_base::beg);     // 30 bytes beyond the beginning
fin.seekg(-1, ios_base::cur);     // back up one byte
fin.seekg(0, ios_base::end);      // go to the end of the file
```

下面来看第二个原型。streampos 类型的值定位到文件中的一个位置。它可以是类，但如果是这样的话，这个类将包含一个接受 streamoff 参数的构造函数和一个接受整数参数的构造函数，以便将两种类型转换为 streampos 值。streampos 值表示文件中的绝对位置（从文件开始处算起）。可以将 streampos 位置看作是相对于文件开始处的位置（以字节为单位，第一个字节的编号为 0）。因此下面的语句将文件指针指向第 112 个字节，这是文件中的第 113 个字节：

```
fin.seekg(112);
```

如果要检查文件指针的当前位置，则对于输入流，可以使用 tellg( )方法，对于输出流，可以使用 tellp( )方法。它们都返回一个表示当前位置的 streampos 值（以字节为单位，从文件开始处算起）。创建 fstream 对象时，输入指针和输出指针将一前一后地移动，因此 tellg( )和 tellp( )返回的值相同。然而，如果使用 istream 对象来管理输入流，而使用 ostream 对象来管理同一个文件的输出流，则输入指针和输出指针将彼此独立地移动，因此 tellg( )和 tellp( )将返回不同的值。

然后，可以使用 seekg( )移到文件的开头。下面是打开文件、移到文件开头并显示文件内容的代码片段：

```
fstream finout;      // read and write streams
finout.open(file,ios::in | ios::out |ios::binary);
//NOTE: Some Unix systems require omitting | ios::binary
int ct = 0;
if (finout.is_open())
{
    finout.seekg(0);     // go to beginning
    cout << "Here are the current contents of the "
          << file << " file:\n";
    while (finout.read((char *) &pl, sizeof pl))
    {
        cout << ct++ << ": " << setw(LIM) << pl.name << ": "
        << setprecision(0) << setw(12) << pl.population
        << setprecision(2) << setw(6) << pl.g << endl;
    }
    if (finout.eof())
        finout.clear(); // clear eof flag
    else
    {
        cerr << "Error in reading " << file << ".\n";
        exit(EXIT_FAILURE);
    }
}
else
{
    cerr << file << " could not be opened -- bye.\n";
    exit(EXIT_FAILURE);
}
```

这与程序清单 17.19 的开头很相似，但也修改和添加了一些内容。首先，程序以读/写模式使用一个 fstream 对象，并使用 seekg( )将文件指针放在文件开头（对于这个例子而言，这其实不是必须的，但它说明了如何使用 seekg( )）。接下来，程序在给记录编号方面做了一些小的改动。然后添加了以下重要的代码：

```
if (finout.eof())
    finout.clear(); // clear eof flag
else
{
    cerr << "Error in reading " << file << ".\n";
    exit(EXIT_FAILURE);
}
```

上述代码解决的问题是，程序读取并显示整个文件后，将设置 eofbit 元素。这使程序相信，它已经处理完文件，并禁止对文件做进一步的读写。使用 clear( )方法重置流状态，并打开 eofbit 后，程序便可以再次访问该文件。else 部分处理程序因到达文件尾之外的其他原因（如硬件故障）而停止读取的情况。

接下来需要确定要修改的记录，并修改它。为此，程序让用户输入记录号。将该编号与记录包含的字节数相乘，得到该记录第一个字节的编号。如果 record 是记录号，则字节编号为 record * sizeof pl：

```
cout << "Enter the record number you wish to change: ";
long rec;
cin >> rec;
eatline();              // get rid of newline
if (rec < 0 || rec >= ct)
{
    cerr << "Invalid record number -- bye\n";
    exit(EXIT_FAILURE);
}
streampos place = rec * sizeof pl;  // convert to streampos type
finout.seekg(place);    // random access
```

变量 ct 表示记录号。如果试图超出文件尾，程序将退出。

接下来，程序显示当前的记录：

```
finout.read((char *) &pl, sizeof pl);
cout << "Your selection:\n";
cout << rec << ": " << setw(LIM) << pl.name << ": "
<< setprecision(0) << setw(12) << pl.population
<< setprecision(2) << setw(6) << pl.g << endl;
if (finout.eof())
    finout.clear();     // clear eof flag
```

显示记录后，程序让您修改记录：

```
cout << "Enter planet name: ";
cin.get(pl.name, LIM);
eatline();
cout << "Enter planetary population: ";
cin >> pl.population;
cout << "Enter planet's acceleration of gravity: ";
cin >> pl.g;
finout.seekp(place);    // go back
finout.write((char *) &pl, sizeof pl) << flush;

if (finout.fail())
{
    cerr << "Error on attempted write\n";
    exit(EXIT_FAILURE);
}
```

程序刷新输出，以确保进入下一步之前，文件被更新。

最后，为显示修改后的文件，程序使用 seekg( )将文件指针重新指向开头。程序清单 17.20 列出了完整

的程序。不要忘了，该程序假设 binary.cpp 创建的 planets.dat 文件是可用的。

注意：实现越旧，与 C++标准相冲突的可能性越大。一些系统不能识别二进制标记、fixed 和 right 控制符以及 ios_base。

程序清单 17.20　random.cpp

```
// random.cpp -- random access to a binary file
#include <iostream>     // not required by most systems
#include <fstream>
#include <iomanip>
#include <cstdlib>      // for exit()
const int LIM = 20;
struct planet
{
    char name[LIM];     // name of planet
    double population;  // its population
    double g;           // its acceleration of gravity
};

const char * file = "planets.dat";  // ASSUMED TO EXIST (binary.cpp example)
inline void eatline() { while (std::cin.get() != '\n') continue; }

int main()
{
    using namespace std;
    planet pl;
    cout << fixed;

// show initial contents
    fstream finout;     // read and write streams
    finout.open(file,
           ios_base::in | ios_base::out | ios_base::binary);
    //NOTE: Some Unix systems require omitting | ios::binary
    int ct = 0;
    if (finout.is_open())
    {
        finout.seekg(0);    // go to beginning
        cout << "Here are the current contents of the "
             << file << " file:\n";
        while (finout.read((char *) &pl, sizeof pl))
        {
            cout << ct++ << ": " << setw(LIM) << pl.name << ": "
                 << setprecision(0) << setw(12) << pl.population
                 << setprecision(2) << setw(6) << pl.g << endl;
        }
        if (finout.eof())
            finout.clear(); // clear eof flag
        else
        {
            cerr << "Error in reading " << file << ".\n";
            exit(EXIT_FAILURE);
        }
    }
    else
```

```
    {
        cerr << file << " could not be opened -- bye.\n";
        exit(EXIT_FAILURE);
    }

// change a record
    cout << "Enter the record number you wish to change: ";
    long rec;
    cin >> rec;
    eatline();              // get rid of newline
    if (rec < 0 || rec >= ct)
    {
        cerr << "Invalid record number -- bye\n";
        exit(EXIT_FAILURE);
    }
    streampos place = rec * sizeof pl;  // convert to streampos type
    finout.seekg(place);    // random access
    if (finout.fail())
    {
        cerr << "Error on attempted seek\n";
        exit(EXIT_FAILURE);
    }

    finout.read((char *) &pl, sizeof pl);
    cout << "Your selection:\n";
    cout << rec << ": " << setw(LIM) << pl.name << ": "
         << setprecision(0) << setw(12) << pl.population
         << setprecision(2) << setw(6) << pl.g << endl;
    if (finout.eof())
        finout.clear();     // clear eof flag

    cout << "Enter planet name: ";
    cin.get(pl.name, LIM);
    eatline();
    cout << "Enter planetary population: ";
    cin >> pl.population;
    cout << "Enter planet's acceleration of gravity: ";
    cin >> pl.g;
    finout.seekp(place);    // go back
    finout.write((char *) &pl, sizeof pl) << flush;
    if (finout.fail())
    {
        cerr << "Error on attempted write\n";
        exit(EXIT_FAILURE);
    }

// show revised file
    ct = 0;
    finout.seekg(0);            // go to beginning of file
    cout << "Here are the new contents of the " << file
         << " file:\n";
    while (finout.read((char *) &pl, sizeof pl))
    {
        cout << ct++ << ": " << setw(LIM) << pl.name << ": "
             << setprecision(0) << setw(12) << pl.population
```

```
            << setprecision(2) << setw(6) << pl.g << endl;
    }
    finout.close();
    cout << "Done.\n";
    return 0;
}
```

下面是程序清单 17.20 中的程序基于 planets.dat 文件的运行情况，该文件比上次见到时多了一些条目：

```
Here are the current contents of the planets.dat file:
0:                Earth:   6928198253  9.81
1:        Jenny's World:     32155648  8.93
2:               Tramtor:  89000000000 15.03
3:               Trellan:      5214000  9.62
4:             Freestone:   3945851000  8.68
5:             Taanagoot:    361000004 10.23
6:                 Marin:       252409  9.79
Enter the record number you wish to change: 2
Your selection:
2:               Tramtor:  89000000000 15.03
Enter planet name: Trantor
Enter planetary population: 89521844777
Enter planet's acceleration of gravity: 10.53
Here are the new contents of the planets.dat file:
0:                Earth:   6928198253  9.81
1:        Jenny's World:     32155648  8.93
2:               Trantor:  89521844777 10.53
3:               Trellan:      5214000  9.62
4:             Freestone:   3945851000  8.68
5:             Taanagoot:    361000004 10.23
6:                 Marin:       252409  9.79
Done.
```

通过使用该程序中的技术，对其进行扩展，使之能够让用户添加新信息和删除记录。如果打算扩展该程序，最好通过使用类和函数来重新组织它。例如，可以将 planet 结构转换为一个类定义，然后对<<插入运算符进行重载，使得 cout<<pl 按示例的格式显示类的数据成员。另外，该示例没有对输入进行检查，您可以添加代码来检查数值输入是否合适。

使用临时文件

开发应用程序时，经常需要使用临时文件，这种文件的存在是短暂的，必须受程序控制。您是否考虑过，在 C++中如何使用临时文件呢？创建临时文件、复制另一个文件的内容并删除文件其实都很简单。首先，需要为临时文件制定一个命名方案，但如何确保每个文件都被指定了独一无二的文件名呢？cstdio 中声明的 tmpnam( )标准函数可以帮助您。

```
char* tmpnam( char* pszName );
```

tmpnam( )函数创建一个临时文件名，将它放在 pszName 指向的 C-风格字符串中。常量 L_tmpnam 和 TMP_MAX（二者都是在 cstdio 中定义的）限制了文件名包含的字符数以及在确保当前目录中不生成重复文件名的情况下 tmpnam( )可被调用的最多次数。下面是生成 10 个临时文件名的代码。

```
#include <cstdio>
#include <iostream>

int main()
{
    using namespace std;
```

```
    cout << "This system can generate up to " << TMP_MAX
         << " temporary names of up to " << L_tmpnam
         << " characters.\n";
    char pszName[ L_tmpnam ] = {'\0'};
    cout << "Here are ten names:\n";
    for( int i=0; 10 > i; i++ )
    {
        tmpnam( pszName );
        cout << pszName << endl;
    }
    return 0;
}
```

更具体地说，使用 tmpnam( )可以生成 TMP_NAM 个不同的文件名，其中每个文件名包含的字符不超过 L_tmpnam 个。生成什么样的文件名取决于实现，您可以运行该程序，来看看编译器给您生成的文件名。

17.5 内核格式化

iostream 族（family）支持程序与终端之间的 I/O，而 fstream 族使用相同的接口提供程序和文件之间的 I/O。C++库还提供了 sstream 族，它们使用相同的接口提供程序和 string 对象之间的 I/O。也就是说，可以使用于 cout 的 ostream 方法将格式化信息写入到 string 对象中，并使用 istream 方法（如 getline( )）来读取 string 对象中的信息。读取 string 对象中的格式化信息或将格式化信息写入 string 对象中被称为内核格式化（incore formatting）。下面简要地介绍一下这些工具（string 的 sstream 族支持取代了 char 数组的 strstream.h 族支持）。

头文件 sstream 定义了一个从 ostream 类派生而来的 ostringstream 类（还有一个基于 wostream 的 wostringstream 类，这个类用于宽字符集）。如果创建了一个 ostringstream 对象，则可以将信息写入其中，它将存储这些信息。可以将可用于 cout 的方法用于 ostringstream 对象。也就是说，可以这样做：

```
ostringstream outstr;
double price = 380.0;
char * ps = " for a copy of the ISO/EIC C++ standard!";
outstr.precision(2);
outstr << fixed;
outstr << "Pay only CHF " << price << ps << endl;
```

格式化文本进入缓冲区，在需要的情况下，该对象将使用动态内存分配来增大缓冲区。ostringstream 类有一个名为 str( )的成员函数，该函数返回一个被初始化为缓冲区内容的字符串对象：

```
string mesg = outstr.str();    // returns string with formatted information
```

使用 str( )方法可以“冻结”该对象，这样便不能将信息写入该对象中。

程序清单 17.21 是一个有关内核格式化的简短示例。

程序清单 17.21　strout.cpp

```
// strout.cpp -- incore formatting (output)
#include <iostream>
#include <sstream>
#include <string>
int main()
{
    using namespace std;
    ostringstream outstr;   // manages a string stream
```

```
    string hdisk;
    cout << "What's the name of your hard disk? ";
    getline(cin, hdisk);
    int cap;
    cout << "What's its capacity in GB? ";
    cin >> cap;
    // write formatted information to string stream
    outstr << "The hard disk " << hdisk << " has a capacity of "
           << cap << " gigabytes.\n";
    string result = outstr.str();   // save result
    cout << result;                 // show contents

    return 0;

}
```

下面是程序清单 17.21 中程序的运行情况：

```
What's the name of your hard disk? Datarapture
What's its capacity in GB? 2000
The hard disk Datarapture has a capacity of 2000 gigabytes.
```

istringstream 类允许使用 istream 方法族读取 istringstream 对象中的数据，istringstream 对象可以使用 string 对象进行初始化。

假设 facts 是一个 string 对象，则要创建与该字符串相关联的 istringstream 对象，可以这样做：

```
istringstream instr(facts);     // use facts to initialize stream
```

这样，便可以使用 istream 方法读取 instr 中的数据。例如，如果 instr 包含大量字符格式的整数，则可以这样读取它们：

```
int n;
int sum = 0;
while (instr >> n)
    sum += n;
```

程序清单 17.22 使用重载的>>运算符读取字符串中的内容，每次读取一个单词。

程序清单 17.22　strin.cpp

```
// strin.cpp -- formatted reading from a char array
#include <iostream>
#include <sstream>
#include <string>
int main()
{
    using namespace std;
    string lit = "It was a dark and stormy day, and "
                 " the full moon glowed brilliantly. ";
    istringstream instr(lit);   // use buf for input
    string word;
    while (instr >> word)       // read a word a time
        cout << word << endl;
    return 0;

}
```

下面是程序清单 17.22 中程序的输出：

```
It
was
a
dark
and
stormy
day,
and
the
full
moon
glowed
brilliantly.
```

总之，istringstream 和 ostringstream 类使得能够使用 istream 和 ostream 类的方法来管理存储在字符串中的字符数据。

17.6 总结

流是进出程序的字节流。缓冲区是内存中的临时存储区域，是程序与文件或其他 I/O 设备之间的桥梁。信息在缓冲区和文件之间传输时，将使用设备（如磁盘驱动器）处理效率最高的尺寸以大块数据的方式进行传输。信息在缓冲区和程序之间传输时，是逐字节传输的，这种方式对于程序中的处理操作更为方便。C++通过将一个被缓冲流同程序及其输入源相连来处理输入。同样，C++也通过将一个被缓冲流与程序及其输出目标相连来处理输出。iostream 和 fstream 文件构成了 I/O 类库，该类库定义了大量用于管理流的类。包含了 iostream 文件的 C++程序将自动打开 8 个流，并使用 8 个对象管理它们。cin 对象管理标准输入流，后者默认与标准输入设备（通常为键盘）相连；cout 对象管理标准输出流，后者默认与标准输出设备（通常为显示器）相连；cerr 和 clog 对象管理与标准错误设备（通常为显示器）相连的未被缓冲的流和被缓冲的流。这 4 个对象有都有用于宽字符的副本，它们是 wcin、wcout、wcerr 和 wclog。

I/O 类库提供了大量有用的方法。istream 类定义了多个版本的抽取运算符（>>），用于识别所有基本的 C++类型，并将字符输入转换为这些类型。get( )方法族和 getline( )方法为单字符输入和字符串输入提供了进一步的支持。同样，ostream 类定义了多个版本的插入运算符（<<），用于识别所有的 C++基本类型，并将它们转换为相应的字符输出。put( )方法对单字符输出提供了进一步的支持。wistream 和 wostream 类对宽字符提供了类似的支持。

使用 ios_base 类方法以及文件 iostream 和 iomanip 中定义的控制符（可与插入运算符拼接的函数），可以控制程序如何格式化输出。这些方法和控制符使得能够控制计数系统、字段宽度、小数位数、显示浮点变量时采用的计数系统以及其他元素。

fstream 文件提供了将 iostream 方法扩展到文件 I/O 的类定义。ifstream 类是从 istream 类派生而来的。通过将 ifstream 对象与文件关联起来，可以使用所有的 istream 方法来读取文件。同样，通过将 ofstream 对象与文件关联起来，可以使用 ostream 方法来写文件；通过将 fstream 对象与文件关联起来，可以将输入和输出方法用于文件。

要将文件与流关联起来，可以在初始化文件流对象时提供文件名，也可以先创建一个文件流对象，然后用 open( )方法将这个流与文件关联起来。close( )方法终止流与文件之间的连接。类构造函数和 open( )方法接受可选的第二个参数，该参数提供文件模式。文件模式决定文件是否被读和/或写、打开文件以便写入时是否截短文件、试图打开不存在的文件时是否会导致错误、是使用二进制模式还是文本模式等。

文本文件以字符格式存储所有的信息，例如，数字值将被转换为字符表示。常规的插入和抽取运算符以及 get( )和 getline( )都支持这种模式。二进制文件使用计算机内部使用的二进制表示来存储信息。与文本文件相比，二进制文件存储数据（尤其是浮点值）更为精确、简洁，但可移植性较差。read( )和 write( )方

法都支持二进制输入和输出。

seekg( )和 seekp( )函数提供对文件的随机存取。这些类方法使得能够将文件指针放置到相对于文件开头、文件尾和当前位置的某个位置。tellg( )和 tellp( )方法报告当前的文件位置。

sstream 头文件定义了 istringstream 和 ostringstream 类，这些类使得能够使用 istream 和 ostream 方法来抽取字符串中的信息，并对要放入到字符串中的信息进行格式化。

17.7　复习题

1．iostream 文件在 C++ I/O 中扮演何种角色？

2．为什么键入数字（如 121）作为输入要求程序进行转换？

3．标准输出与标准错误之间有什么区别？

4．为什么在不为每个类型提供明确指示的情况下，cout 仍能够显示不同的 C++类型？

5．输出方法的定义的哪一特征让您能够拼接输出？

6．编写一个程序，要求用户输入一个整数，然后以十进制、八进制和十六进制显示该整数。在宽度为 15 个字符的字段中显示每种形式，并将它们显示在同一行上，同时使用 C++数基前缀。

7．编写一个程序，请求用户输入下面的信息，并按下面的格式显示它们：

```
Enter your name: Billy Gruff
Enter your hourly wages: 12
Enter number of hours worked: 7.5
First format:
                  Billy Gruff: $     12.00:  7.5
Second format:
Billy Gruff                    : $12.00     :7.5
```

8．对于下面的程序：

```
//rq17-8.cpp
#include <iostream>

int main()
{
    using namespace std;
    char ch;
    int ct1 = 0;

    cin >> ch;
    while (ch != 'q')
    {
        ct1++;
        cin >> ch;
    }

    int ct2 = 0;
    cin.get(ch);
    while (ch != 'q')
    {
        ct2++;
        cin.get(ch);
    }
    cout << "ct1 = " << ct1 << "; ct2 = " << ct2 << "\n";
```

```
    return 0;
}
```

如果输入如下，该程序将打印什么内容？

```
I see a q<Enter>
I see a q<Enter>
```

其中，<Enter>表示按回车键。

9．下面的两条语句都读取并丢弃行尾之前的所有字符（包括行尾）。这两条语句的行为在哪方面不同？

```
while (cin.get() != '\n')
    continue;
cin.ignore(80, '\n');
```

17.8 编程练习

1．编写一个程序计算输入流中第一个$之前的字符数目，并将$留在输入流中。

2．编写一个程序，将键盘输入（直到模拟的文件尾）复制到通过命令行指定的文件中。

3．编写一个程序，将一个文件复制到另一个文件中。让程序通过命令行获取文件名。如果文件无法打开，程序将指出这一点。

4．编写一个程序，它打开两个文本文件进行输入，打开一个文本文件进行输出。该程序将两个输入文件中对应的行并接起来，并用空格分隔，然后将结果写入到输出文件中。如果一个文件比另一个短，则将较长文件中余下的几行直接复制到输出文件中。例如，假设第一个输入文件的内容如下：

```
eggs kites donuts
balloons hammers
stones
```

而第二个输入文件的内容如下：

```
zero lassitude
finance drama
```

则得到的文件的内容将如下：

```
eggs kites donuts zero lassitude
balloons hammers finance drama
stones
```

5．Mat 和 Pat 想邀请他们的朋友来参加派对，就像第 16 章中的编程练习 8 那样，但现在他们希望程序使用文件。他们请您编写一个完成下述任务的程序。

- 从文本文件 mat.dat 中读取 Mat 朋友的姓名清单，其中每行为一个朋友。姓名将被存储在容器，然后按顺序显示出来。
- 从文本文件 pat.dat 中读取 Pat 朋友的姓名清单，其中每行为一个朋友。姓名将被存储在容器中，然后按顺序显示出来。
- 合并两个清单，删除重复的条目，并将结果保存在文件 matnpat.dat 中，其中每行为一个朋友。

6．考虑 14 章的编程练习 5 中的类定义。如果还没有完成这个练习，请现在就做，然后完成下面的任务。

编写一个程序，它使用标准 C++ I/O、文件 I/O 以及 14 章的编程练习 5 中定义的 employee、manager、fink 和 highfink 类型的数据。该程序应包含程序清单 17.17 中的代码行，即允许用户将新数据添加到文件中。该程序首次被运行时，将要求用户输入数据，然后显示所有的数据，并将这些信息保存到一个文件中。当该程序再次被运行时，将首先读取并显示文件中的数据，然后让用户添加数据，并显示所有的数据。差别之一是，应通过一个指向 employee 类型的指针数组来处理数据。这样，指针可以指向 employee 对象，

也可以指向从 employee 派生出来的其他三种对象中的任何一种。使数组较小有助于检查程序，例如，您可能将数组限定为最多包含 10 个元素：

```
const int MAX = 10;     // no more than 10 objects
...
employee * pc[MAX];
```

为通过键盘输入，程序应使用一个菜单，让用户选择要创建的对象类型。菜单将使用一个 switch，以便使用 new 来创建指定类型的对象，并将它的地址赋给 pc 数组中的一个指针。然后该对象可以使用虚函数 setall( )来提示用户输入相应的数据：

```
pc[i]->setall();  // invokes function corresponding to type of object
```

为将数据保存到文件中，应设计一个虚函数 writeall( )：

```
for (i = 0; i < index; i++)
    pc[i]->writeall(fout);// fout ofstream connected to output file
```

注意：对于这个练习，应使用文本 I/O，而不是二进制 I/O（遗憾的是，虚对象包含指向虚函数指针表的指针，而 write( )将把这种信息复制到文件中。使用 read( )读取文件的内容，以填充对象时，函数指针值将为乱码，这将扰乱虚函数的行为）。可使用换行符将字段分隔开，这样在输入时将很容易识别各个字段。也可以使用二进制 I/O，但不能将对象作为一个整体写入，而应该提供分别对每个类成员应用 write( )和 read( )的类方法。这样，程序将只把所需的数据保存到文件中。

比较难处理的部分是使用文件恢复数据。问题在于：程序如何才能知道接下来要恢复的项目是 employee 对象、manager 对象、fink 对象还是 highfink 对象？一种方法是，在对象的数据写入文件时，在数据前面加上一个指示对象类型的整数。这样，在文件输入时，程序便可以读取该整数，并使用 switch 语句创建一个适当的对象来接收数据：

```
enum classkind{Employee, Manager, Fink, Highfink}; // in class header
...
int classtype;
while((fin >> classtype).get(ch)){ // newline separates int from data
    switch(classtype) {
        case Employee   : pc[i] = new employee;
                        : break;
```

然后便可以使用指针调用虚函数 getall( )来读取信息：

```
pc[i++]->getall();
```

7. 下面是某个程序的部分代码。该程序将键盘输入读取到一个由 string 对象组成的 vector 中，将字符串内容（而不是 string 对象）存储到一个文件中，然后该文件的内容复制到另一个由 string 对象组成的 vector 中。

```
int main()
{
    using namespace std;
    vector<string> vostr;
    string temp;

// acquire strings
    cout << "Enter strings (empty line to quit):\n";
    while (getline(cin,temp) && temp[0] != '\0')
        vostr.push_back(temp);
    cout << "Here is your input.\n";
    for_each(vostr.begin(), vostr.end(), ShowStr);
```

```
// store in a file
    ofstream fout("strings.dat", ios_base::out | ios_base::binary);
    for_each(vostr.begin(), vostr.end(), Store(fout));
    fout.close();

// recover file contents
    vector<string> vistr;
    ifstream fin("strings.dat", ios_base::in | ios_base::binary);
    if (!fin.is_open())
    {
        cerr << "Could not open file for input.\n";
        exit(EXIT_FAILURE);
    }
    GetStrs(fin, vistr);
    cout << "\nHere are the strings read from the file:\n";
    for_each(vistr.begin(), vistr.end(), ShowStr);

    return 0;
}
```

该程序以二进制格式打开文件，并想使用 read( )和 write( )来完成 I/O。余下的工作如下所述。

- 编写函数 void ShowStr(const string &)，它显示一个 string 对象，并在显示完后换行。
- 编写函数符 Store，它将字符串信息写入到文件中。Store 的构造函数应接受一个指定 ifstream 对象的参数，而重载的 operator( )(const string &)应指出要写入到文件中的字符串。一种可行的计划是，首先将字符串的长度写入到文件中，然后将字符串的内容写入到文件中。例如，如果 len 存储了字符串的长度，可以这样做：

```
os.write((char *)&len, sizeof(std::size_t));  // store length
os.write(s.data(), len);                      // store characters
```

  成员函数 data( )返回一个指针，该指针指向一个其中存储了字符串中字符的数组。它类似于成员函数 c_str( )，只是后者在数组末尾加上了一个空字符。
- 编写函数 GetStrs( )，它根据文件恢复信息。该函数可以使用 read( )来获得字符串的长度，然后使用一个循环从文件中读取相应数量的字符，并将它们附加到一个原来为空的临时 string 末尾。由于 string 的数据是私有的，因此必须使用 string 类的方法来将数据存储到 string 对象中，而不能直接存储。

第 18 章　探讨 C++新标准

本章首先复习前面介绍过的 C++11 功能，然后介绍如下主题：

- 移动语义和右值引用。
- Lambda 表达式。
- 包装器模板 function。
- 可变参数模板。

本章重点介绍 C++11 对 C++所做的改进。本书前面介绍过多项 C++11 功能，本章首先复习这些功能，并详细介绍其他一些功能。然后，指出一些超出了本书范围的 C++11 新增功能（考虑到 C++11 草案的篇幅比 C++98 长 98%，本书无法全面介绍）。最后，将简要地探讨 BOOST 库。

18.1　复习前面介绍过的 C++11 功能

本书前面介绍过很多 C++11 改进，但您现在可能忘了，本节简要地复习这些改进。

18.1.1　新类型

C++11 新增了类型 long long 和 unsigned long long，以支持 64 位（或更宽）的整型；新增了类型 char16_t 和 char32_t，以支持 16 位和 32 位的字符表示；还新增了“原始”字符串。第 3 章讨论了这些新增的类型。

18.8.2　统一的初始化

C++11 扩大了用大括号括起的列表（初始化列表）的适用范围，使其可用于所有内置类型和用户定义的类型（即类对象）。使用初始化列表时，可添加等号（=），也可不添加：

```
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：

```
#include <initializer_list>
double sum(std::initializer_list<double> il);
int main()
{
   double total = sum({2.5,3.1,4});  // 4 converted to 4.0
...
}
double sum(std::initializer_list<double> il)
{
    double tot = 0;
    for (auto p = il.begin(); p !=il.end(); p++)
        tot += *p;
    return tot;
}
```

18.1.3 声明

C++11 提供了多种简化声明的功能，尤其在使用模板时。

1. auto

以前，关键字 auto 是一个存储类型说明符（见第 9 章），C++11 将其用于实现自动类型推断（见第 3 章）。这要求进行显式初始化，让编译器能够将变量的类型设置为初始值的类型：

```
auto maton = 112;  // maton is type int
auto pt = &maton;  // pt is type int *
double fm(double, int);
auto pf = fm;      // pf is type double (*)(double,int)
```

关键字 auto 还可简化模板声明。例如，如果 il 是一个 std::initializer_list<double>对象，则可将下述代码：

```
for (std::initializer_list<double>::iterator p = il.begin();
                                              p !=il.end(); p++)
```

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 T*U 的类型，这里假定定义了运算 T*U。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，但如果您试图将 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: xxxxxxxxxx
object two: xxxxxxxxxx
object three: oooooooooooooooooooo
object four: xxxxxxxxxxoooooooooooooooooooo
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: xxxxxxxxxx
object two: xxxxxxxxxx
object three: oooooooooooooooooooo
object four: xxxxxxxxxxoooooooooooooooooooo
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: xxxxxxxxxx
object two: xxxxxxxxxxxxxxxxxxxx
three = one
copy assignment operator called:
```

```
now object three = xxxxxxxxxx
and object one = xxxxxxxxxx
four = one + two
move assignment operator called:
now object four = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
four = move(one)
move assignment operator called:
now object four = xxxxxxxxxx
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。如果也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) {}
    B0(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，多个少整数可被 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 decltyp 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，lambad 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 lambad 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 bind2ed，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、men_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。这里的核心理念是，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），必须写成字符字面量“\\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 Dsign 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 用户的经验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。

附录 A　计数系统

人们使用很多计数系统来表示数字。有些计数系统（如罗马数字）不适合用于算术运算；而印度计数系统经过改进，并传入到欧洲后变成了阿拉伯计数系统，这种数字方便了数学、科学和商业计算。现代的计算机计数系统是基于占位符概念的，使用了最先出现在印度计数系统中的零。然而，这种原理被推广到其他计数系统。因此，虽然在日常生活中使用的是下一节将介绍的十进制，但计算领域通常使用八进制、十六进制和二进制。

A.1　十进制数

数字的书写方式是基于 10 的幂数。例如，对于数字 2468，2 表示 2 个 1000，4 表示 4 个 100，6 表示 6 个 10，8 表示 8 个 1。

$2468=2\times1000+4\times100+6\times10+8\times1$

一千是 $10\times10\times10$ 或 10 的 3 次幂，用 10^3 表示。使用这种表示法，可以这样书写上述关系：

$2468=2\times10^3+4\times10^2+6\times10^1+8\times10^0$

因为这种数字表示法是基于 10 的幂，所以将它称作基数为 10 的表示法或十进制表示法。可以用任何数作基数。例如，C++允许使用基数 8（八进制）和基数 16（十六进制）来书写整数（请注意，10^0 为 1，任何非零数的 0 次幂都为 1）。

A.2　八进制整数

八进制数是基于 8 的幂的，所以基数为 8 的表示法用数字 0-7 来书写数字。C++用前缀 0 来表示八进制表示法。也就是说，0177 是一个八进制值。可以用 8 的幂来找到对应的十进制值：

八　进　制	十　进　制
177	$=1\times8^2+7\times8^1+7\times8^0$
	$=1\times64+7\times8+7\times1$
	$=127$

由于 UNIX 操作系统常使用八进制来表示值，因此 C++和 C 提供了八进制表示法。

A.3　十六进制数

十六进制数是基于 16 的幂的。这意味着十六进制的 10 表示 16+0，即 16。为表示 9-16 值，需要其他一些数字，标准的十六进制表示法使用字母 a-f。C++接受这些字符的大写和小写版本，如表 A.1 所示。

表 A.1 十六进制数

十六进制数	十 进 制 值
a 或 A	10
b 或 B	11
c 或 C	12
d 或 D	13
e 或 E	14
f 或 F	15

C++使用 0x 或 0X 来指示十六进制表示法。因此 0x2B3 是一个十六进制值，可使用 16 的幂来得到对应的十进制值。

十六进制	十 进 制
0x2B3	$=2\times16^2+11\times16^1+3\times16^0$
	$=2\times256+11\times16+3\times1$
	$=691$

硬件文档常使用十六进制来表示诸如内存单元和端口号等值。

A.4 二进制数

不管是使用十进制、八进制，还是十六进制表示法来书写整数，计算机都将它存储为二进制值（即基数为 2）。二进制表示法只使用两个数字——0 和 1。例如，10011011 就是二进制数。但 C++没有提供二进制表示法来书写数字的方式。二进制数是基于 2 的幂。

二 进 制	十 进 制
10011011	$=1\times2^7+0\times2^6+0\times2^5+1\times2^4+1\times2^3+0\times2^2+1\times2^1+1\times2^0$
	$=128+0+0+16+8+0+2+1$
	$=155$

二进制表示法与计算机内存完全对应，在内存中，每个单元（位）都可以设置成开或关。只是将关表示为 0，将开表示为 1。内存通常是以字节为单位组织的，每个字节包含 8 位（正如第 2 章指出的，C++字节并非一定是 8 位，但本附录采用常见的做法，用字节表示八位组）。字节中的位被编号，对应于相关的 2 的幂。这样，最右侧的位编号为 0，然后是 1，依此类推。例如，图 A.1 表示一个 2 字节的整数。

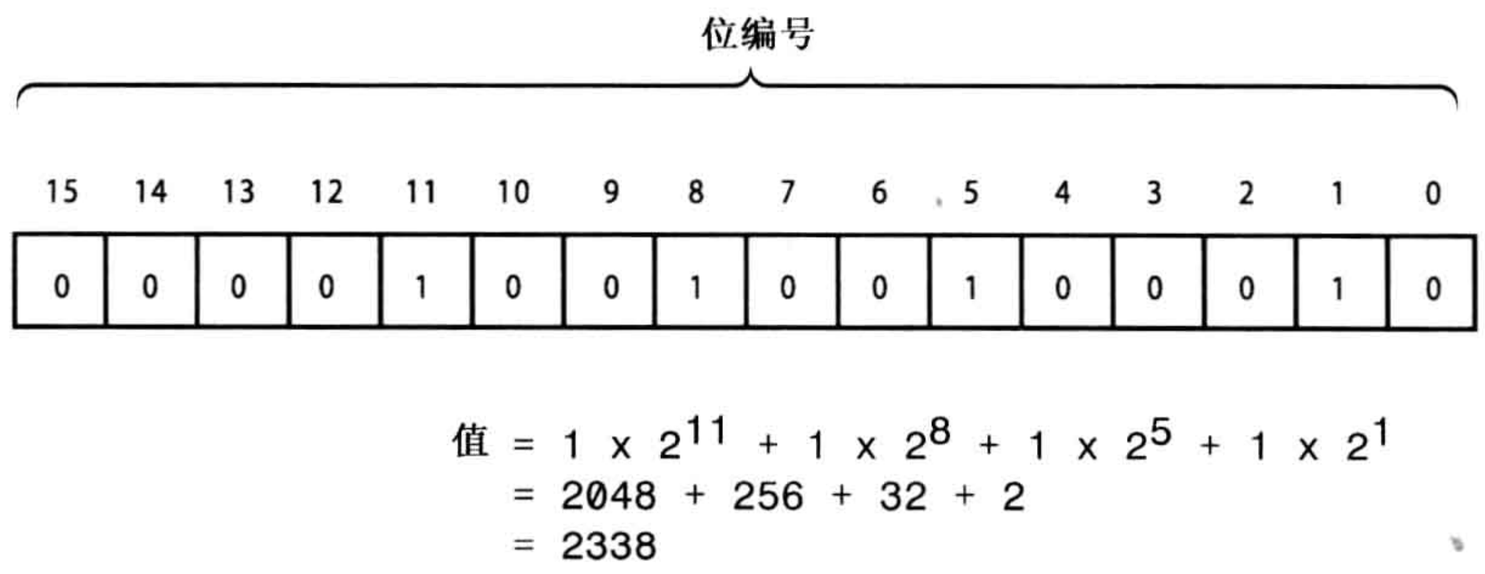

图 A.1 2 字节整数值

A.5 二进制和十六进制

十六进制表示法常用于提供更为方便的二进制数据（如内存地址或存储位标记设置的整数）视图。这样做的原因是，每个十六进制位对应于 4 位。表 A.2 说明了这种对应关系。

表 A.2 十六进制数和对应的二进制数

十六进制位	对应的二进制数
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

要将十六进制值转换为二进制，只需将每个十六进制位替换为相应的二进制数即可。例如，十六进制 0xA4 对应于二进制数 10100100。同样，可以轻松地将二进制值转换为十六进制，方法是将每 4 位转换为对应的十六进制位。例如，二进制值 10010101 将被转换为 0x95。

Big Endian 和 Little Endian

奇怪的是，都使用整数的二进制表示的两个计算平台对同一个值的表示可能并不相同。例如，Intel 计算机使用 Little Endian 体系结构来存储字节，而 Motorola 处理器、IBM 大型机、SPARC 处理器和 ARM 处理器使用 Big Endian 方案（但最后的两种系统可配置成使用上述任何一种方案）。

术语 Big Endian 和 Little Endian 是从“Big End In”和“Little End In”（指内存中单词（通常为两个字节）的字节顺序）衍生而来的。在 Intel 计算机（Little Endian）中，先存储低位字节，这意味着十六进制值 0xABCD 在内存中将被存储为 0xCD 0xAB。Motorola（Big Endian）计算机按相反的顺序存储，因此 0xABCD 在内存中被存储为 0xAB 0xCD。

这些术语最先出现在 Jonathan Swift 编写的《Gulliver’s Travels》一书中。为讽刺众多政治斗争的非理性，Swift 杜撰了假想国中两个喜欢争论的政治派别：Big Endians 和 Little Endians，前者坚持认为从大的一头打破鸡蛋更合理，而后者坚持认为从小的一头打破鸡蛋更合理。

作为软件工程师，应了解目标平台的词序，它会影响对通过网络传输的数据的解释方式以及数据在二进制文件中的存储方式。在上面的例子中，二字节内存模式 0xABCD 在 Little Endian 计算机上表示十进制值 52651，而在 Big Endian 计算机上表示十进制值 43981。

附录 B　C++保留字

C++保留了一些单词供自己和 C++库使用。程序员不应将保留字用作声明中的标识符。保留字分三类：关键字、替代标记（alternative token）和 C++库保留名称。

B.1　C++关键字

关键字是组成编程语言词汇表的标识符，它们不能用于其他用途，如用作变量名。表 B.1 列出了 C++关键字，其中以粗体显示的关键字也是 ANSI C99 标准中的关键字，而以斜体显示的关键字是 C++11 新增的。

表 B.1　　C++关键字

alignas	alignof	asm	**auto**	bool
break	**case**	catch	**char**	char16_t
char32_t	class	**const**	const_cast	constexpr
continue	decltype	**default**	delete	**do**
double	dynamic_cast	**else**	**enum**	explicit
export	**extern**	false	**float**	**for**
friend	**goto**	**if**	**inline**	**int**
long	mutable	namespace	new	noexcept
nullptr	operator	private	protected	public
register	reinterpret_cast	**return**	**short**	**signed**
sizeof	**static**	static_assert	static_cast	**struct**
switch	template	this	thread_local	throw
true	try	**typedef**	typeid	typename
union	**unsigned**	using	virtual	**void**
volatile	wchar_t	**while**		

B.2　替代标记

除关键字外，C++还有一些运算符的字母替代表示，它们被称为替代标记。替代标记也被保留，表 B.2 列出了替代标记及其表示的运算符。

表 B.2　　C++保留的替代标记及其含义

标　记	含　义
and	&&
and_eq	&=

续表

标　　记	含　　义
bitand	&
bitor	|
compl	~
not	!
not_eq	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

B.3　C++库保留名称

编译器不允许程序员将关键字和替代标记用作名称。还有另一类禁止使用（但并非绝对不能用）的名称——保留名称，它们是保留给 C++库使用的名称。如果您将这种名称用作标识符，后果将是不确定的。也就是说，可能导致编译器错误、警告、程序不能正确运行或根本不会导致任何问题。

C++语言保留了库头文件中使用的宏名。如果程序包含某个头文件，则不应将该头文件（以及该头文件包含的头文件，依此类推）中定义的宏名用作其他目的。例如，如果您直接或间接地包含了头文件<climits>，则不应将 CHAR_BIT 用作标识符，因为它已被用作该头文件中一个宏的名称。

C++语言保留了以两个下划线或下划线和大写字母打头的名称，还将以单个下划线打头的名称保留用作全局变量。因此，程序员不能在全局名称空间使用诸如__gink、__Lynx 和_lynx 等名称。

C++语言保留了在库头文件中被声明为链接性为外部的名称。对于函数，这包括函数的特征标（名称和参数列表）。例如，假设有如下代码：

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

则函数特征标 tan（double）被保留。这意味着您的程序不应声明一个原型如下所示的函数：

```
int tan(double);  // don't do it
```

该原型确实与库函数 tan( ) 的原型不同，因为后者的返回类型为 double，但特征标部分确实相同。然而，定义下面的原型是可以的：

```
char * tan(char *);  // ok
```

这是因为虽然其名称与库函数 tan( )相同，但特征标不同。

B.4　有特殊含义的标识符

C++社区讨厌新增关键字，因为它们可能与现有代码发生冲突。这就是标准委员会改变关键字 auto 的用法，并赋予其他关键字（如 virtual 和 delete）新用法的原因所在。C++11 提供了另一种避免新增关键字的机制，即使用具有特殊含义的标识符。这些标识符不是关键字，但用于实现语言功能。编译器根据上下文来判断它们是常规标识符还是用于实现语言功能：

```
class F
{
    int final;                          // #1
public:
...
    virtual void unfold() {...} = final;  // #2
};
```

在上述代码中，语句#1 中的 final 是一个常规标识符，而语句#2 中的 final 使用了一种语言功能。这两种用法彼此不会冲突。

另外，C++还有很多经常出现在程序中，但不被保留的标识符。这包括头文件名、库函数名和 main（必不可少的函数的名称，程序从该函数开始执行）。只要不发生名称空间冲突，就可将这些标识符用于其他目的，但没有理由这样做。也就是说，完全可以编写下面的代码，但常识告诉您不应这样做：

```
// allowable but silly
#include <iostream>
int iostream(int a);
int main ()
{
    std::
cout << iostream(5) << '\n';
    return 0;
}

int iostream(int a)
{
    int main = a + 1;
    int cout = a -1;
    return main*cout;
}
```

附录 C　ASCII 字符集

计算机使用数字代码来存储字符。ASCII 码是美国最常用的编码，它是 Unicode 的一个子集（一个非常小的子集）。C++使得能够直接表示大多数字符，方法是将字符用单引号括起，例如'A'表示字符 A。也可以用前面带反斜杠的八进制或十六进制编码来表示单个字符，例如，'\012'和'\0xa'表示的都是换行符（LF）。这种转义序列还可放在字符串中，如"Hello, \012my dear"。

表 C.1 列出了以各种方式表示的 ASCII 字符集。在该表中，当被用作前缀时，^字符表示使用 Ctrl 键。

表 C.1　　ASCII 字符集

十 进 制	八 进 制	十 六 进 制	二 进 制	字　符	ASCII 名称
0	0	0	00000000	^@	NUL
1	01	0x1	00000001	^A	SOH
2	02	0x2	00000010	^B	STX
3	03	0x3	00000011	^C	ETX
4	04	0x4	00000100	^D	EOT
5	05	0x5	00000101	^E	ENQ
6	06	0x6	00000110	^F	ACK
7	07	0x7	00000111	^G	BEL
8	010	0x8	00001000	^H	BS
9	011	0x9	00001001	^I，tab	HT
10	012	0xa	00001010	^J	LF
11	013	0xb	00001011	^K	VT
12	014	0xc	00001100	^L	FF
13	015	0xd	00001101	^M	CR
14	016	0xe	00001110	^N	SO
15	017	0xf	00001111	^O	SI
16	020	0x10	00010000	^P	DLE
17	021	0x11	00010001	^Q	DC1
18	022	0x12	00010010	^R	DC2
19	023	0x13	00010011	^S	DC3
20	024	0x14	00010100	^T	DC4
21	025	0x15	00010101	^U	NAK
22	026	0x16	00010110	^V	SYN
23	027	0x17	00010111	^W	ETB
24	030	0x18	00011000	^X	CAN
25	031	0x19	00011001	^Y	EM
26	032	0x1a	00011010	^Z	SUB
27	033	0x1b	00011011	^[，Esc	ESC
28	034	0x1c	00011100	^\	FS

续表

十 进 制	八 进 制	十 六 进 制	二 进 制	字 符	ASCII 名称
29	035	0x1d	00011101	^]	GS
30	036	0x1e	00011110	^^	RS
31	037	0x1f	00011111	^_	US
32	040	0x20	00100000	空格	SP
33	041	0x21	00100001	!	
34	042	0x22	00100010	”	
35	043	0x23	00100011	#	
36	044	0x24	00100100	$	
37	045	0x25	00100101	%	
38	046	0x26	00100110	&	
39	047	0x27	00100111	'	
40	050	0x28	00101000	(	
41	051	0x29	00101001	)	
42	052	0x2a	00101010	*	
43	053	0x2b	00101011	+	
44	054	0x2c	00101100	'	
45	055	0x2d	00101101	-	
46	056	0x2e	00101110	.	
47	057	0x2f	00101111	/	
48	060	0x30	00110000	0	
49	061	0x31	00110001	1	
50	062	0x32	00110010	2	
51	063	0x33	00110011	3	
52	064	0x34	00110100	4	
53	065	0x35	00110101	5	
54	066	0x36	00110110	6	
55	067	0x37	00110111	7	
56	070	0x38	00111000	8	
57	071	0x39	00111001	9	
58	072	0x3a	00111010	:	
59	073	0x3b	00111011	;	
60	074	0x3c	00111100	<	
61	075	0x3d	00111101	=	
62	076	0x3e	00111110	>	
63	077	0x3f	00111111	?	
64	0100	0x40	01000000	@	
65	0101	0x41	01000001	A	
66	0102	0x42	01000010	B	
67	0103	0x43	01000011	C	
68	0104	0x44	01000100	D	
69	0105	0x45	01000101	E	

续表

十　进　制	八　进　制	十六进制	二　进　制	字　　符	ASCII 名称
70	0106	0x46	01000110	F	
71	0107	0x47	01000111	G	
72	0110	0x48	01001000	H	
73	0111	0x49	01001001	I	
74	0112	0x4a	01001010	J	
75	0113	0x4b	01001011	K	
76	0114	0x4c	01001100	L	
77	0115	0x4d	01001101	M	
78	0116	0x4e	01001110	N	
79	0117	0x4f	01001111	O	
80	0120	0x50	01010000	P	
81	0121	0x51	01010001	Q	
82	0122	0x52	01010010	R	
83	0123	0x53	01010011	S	
84	0124	0x54	01010100	T	
85	0125	0x55	01010101	U	
86	0126	0x56	01010110	V	
87	0127	0x57	01010111	W	
88	0130	0x58	01011000	X	
89	0131	0x59	01011001	Y	
90	0132	0x5a	01011010	Z	
91	0133	0x5b	01011011	[	
92	0134	0x5c	01011100	\	
93	0135	0x5d	01011101	]	
94	0136	0x5e	01011110	^	
95	0137	0x5f	01011111	-	
96	0140	0x60	01100000	'	
97	0141	0x61	01100001	a	
98	0142	0x62	01100010	b	
99	0143	0x63	01100011	c	
100	0144	0x64	01100100	d	
101	0145	0x65	01100101	e	
102	0146	0x66	01100110	f	
103	0147	0x67	01100111	g	
104	0150	0x68	01101000	h	
105	0151	0x69	01101001	i	
106	0152	0x6a	01101010	j	
107	0153	0x6b	01101011	k	
108	0154	0x6c	01101100	l	
109	0155	0x6d	01101101	m	
110	0156	0x6e	01101110	n	

续表

十 进 制	八 进 制	十 六 进 制	二 进 制	字 符	ASCII 名称
111	0157	0x6f	01101111	o	
112	0160	0x70	01110000	p	
113	0161	0x71	01110001	q	
114	0162	0x72	01110010	r	
115	0163	0x73	01110011	s	
116	0164	0x74	01110100	t	
117	0165	0x75	01110101	u	
118	0166	0x76	01110110	v	
119	0167	0x77	01110111	w	
120	0170	0x78	01111000	x	
121	0171	0x79	01111001	y	
122	0172	0x7a	01111010	z	
123	0173	0x7b	01111011	{	
124	0174	0x7c	01111100	|	
125	0175	0x7d	01111101	}	
126	0176	0x7e	01111110	~	
127	0177	0x7f	01111111	Del	

附录 D　运算符优先级

运算符优先级决定了运算符用于值的顺序。C++运算符分为 18 个优先级组，如表 D.1 所示。第 1 组中的运算符的优先级最高，第 2 组中运算符的优先级次之，依此类推。如果两个运算符被用于同一个操作数，则首先应用优先级高的运算符。如果两个运算符的优先级相同，则 C++使用结合性规则来决定哪个运算符结合得更为紧密。同一组中运算符的优先级和结合性相同，不管是从左到右（表中 L-R）还是从右到左（表中 R-L）结合。从左到右的结合性意味着首先应用最左边的运算符，而从右到左的结合性则意味着首先应用最右边的运算符。

表 D.1　　C++运算符的优先级和结合性

运　算　符	结　合　性	含　　义
优先级第 1 组		
::		作用域解析运算符
优先级第 2 组		
（表达式）		分组
()	L-R	函数调用
()		值构造，即 type(expr)
[]		数组下标
->		间接成员运算符
.		直接成员运算符
const_cast		专用的类型转换
dynamic_cast		专用的类型转换
reinterpret_cast		专用的类型转换
static_cast		专用的类型转换
typeid		类型标识
++		加 1 运算符，后缀
- -		减 1 运算符，后缀
优先级第 3 组（全是一元运算符）		
!	R-L	逻辑非
~		位非
+		一元加号（正号）
-		一元减号（负号）
++		加 1 运算符，前缀
- -		减 1 运算符，前缀
&		地址
*		解除引用（间接值）
()		类型转换，即(type)expr
sizeof		长度，以字节为单位
new		动态分配内存
new [ ]		动态分配数组
delete		动态释放内存
delete [ ]		动态释放数组

续表

运 算 符	结 合 性	含 义
优先级第 4 组		
.*	L-R	成员解除引用
->*		间接成员解除引用
优先级第 5 组（全是二元运算符）		
*	L-R	乘
/		除
^		模（余数）
优先级第 6 组（全是二元运算符）		
+	L-R	加
-		减
优先级第 7 组		
<<	L-R	左移
>>		右移
优先级第 8 组		
<	L-R	小于
<=		小于或等于
>=		大于或等于
>		大于
优先级第 9 组		
==	L-R	等于
!=		不等于
优先级第 10 组（一元运算符）		
&	L-R	按位 AND
优先级第 11 组		
^	L-R	按位 XOF（异或）
优先级第 12 组		
|	L-R	按位 OR
优先级第 13 组		
&&	L-R	逻辑 AND
优先级第 14 组		
||	L-R	逻辑 OR
优先级第 15 组		
:?	R-L	条件
优先级第 16 组		
=	R-L	简单赋值
*=		乘并赋值
/=		除并赋值
%=		求模并赋值
+=		加并赋值
-=		减并赋值
&=		按位 AND 并赋值
^=		按位 XOR 并赋值
|=		按位 OR 并赋值
<<=		左移并赋值
>>=		右移并赋值

续表

运　算　符	结　合　性	含　　义
优先级第 17 组		
throw	L-R	引发异常
优先级第 18 组		
,	L-R	将两个表达式合并成一个

有些符号（如*或&）被用作多个运算符。在这种情况下，一种形式是一元（一个操作数），另一种形式是二元（两个操作数），编译器将根据上下文来确定使用哪种形式。对于同一个符号可以两种方式使用的情况，表 D.1 将运算符标记为一元组或二元组。

下面介绍一些优先级和结合性的例子。

对于下面的例子，编译器必须决定先将 5 和 3 相加，还是先将 5 和 6 相乘：

```
3 + 5 * 6
```

*运算符的优先级比+运算符高，所以它被首先用于 5，因此表达式变成 3 +30，即 33。

对于下面的例子，编译器必须决定先将 120 除以 6，还是先将 6 和 5 相乘：

```
120 / 6 * 5
```

/和*的优先级相同，但这些运算符从左到右结合的。这意味着首先应用操作数（6）左侧的运算符，因此表达式变为 20*5，即 100。

对于下面的例子，编译器必须决定先对 str 递增还是先对 str 解除引用：

```
char * str = "Whoa";
char ch = *str++;
```

后缀++运算符的优先级比一元运算符*高，这意味着加号运算符将对 str 进行操作，而不是对*str 进行操作。也就是说，将指针加 1，使之指向下一个字符，而不是修改被指针指向的字符。不过，由于++是后缀形式，因此将在将*str 的值赋给 ch 后，再将指针加 1。因此，上述表达式将字符 W 赋给 ch，然后移动指针 str，使之指向字符 h。

下面是一个类似的例子：

```
char * str = "Whoa";
char ch = *++str;
```

前缀++运算符和一元运算符*的优先级相同，但它们是从右到左结合的。因此，str（不是*str）将被加 1。因为++运算符是前缀形式，所以首先将 str 加 1，然后将得到的指针执行解除引用的操作。因此，str 将指向字符 h，并将字符 h 赋给 ch。

注意，表 D.1 在“优先级”行中使用一元或二元来区分使用同一个符号的两个运算符，如一元地址运算符和二元按位 AND 运算符。

附录 B 列出了一些运算符的替代表示。

附录 E　其他运算符

为了避免篇幅过长，有三组运算符没有在本书正文部分介绍。第一组是按位运算符，能够操纵值中的各个位；这些运算符是从 C 语言继承而来的；第二组运算符是两个成员解除引用运算符，它们是 C++新增的；第三组是 C++11 新增的运算符：alignof 和 noexcept。本附录将简要地对这些运算符做一总结。

E.1　按位运算符

按位运算符对整数值的位进行操作。例如，左移运算符将位向左移，按位非运算符将所有的 1 变成 0，所有的 0 变成 1，C++共有 6 个这样的运算符：<<、>>、~、&、|和^。

E.1.1　移位运算符

左移运算符的语法如下：

```
value << shift
```

其中，value 是要被操作的整数值，shift 是要移动的位数。例如，下面的代码将值 13 的所有位都向左移 3 位：

```
13 << 3
```

腾出的位置用 0 填充，超出边界的位被丢弃（参见图 E.1）。

由于每个位都表示右边一位的 2 倍（参见附录 A），所以左移一位相当于乘以 2。同样，左移 2 位相当于乘以 2^2，左移 n 位相当于乘以 2^n。因此，13<<3 的值为 13×2^3，即 104。

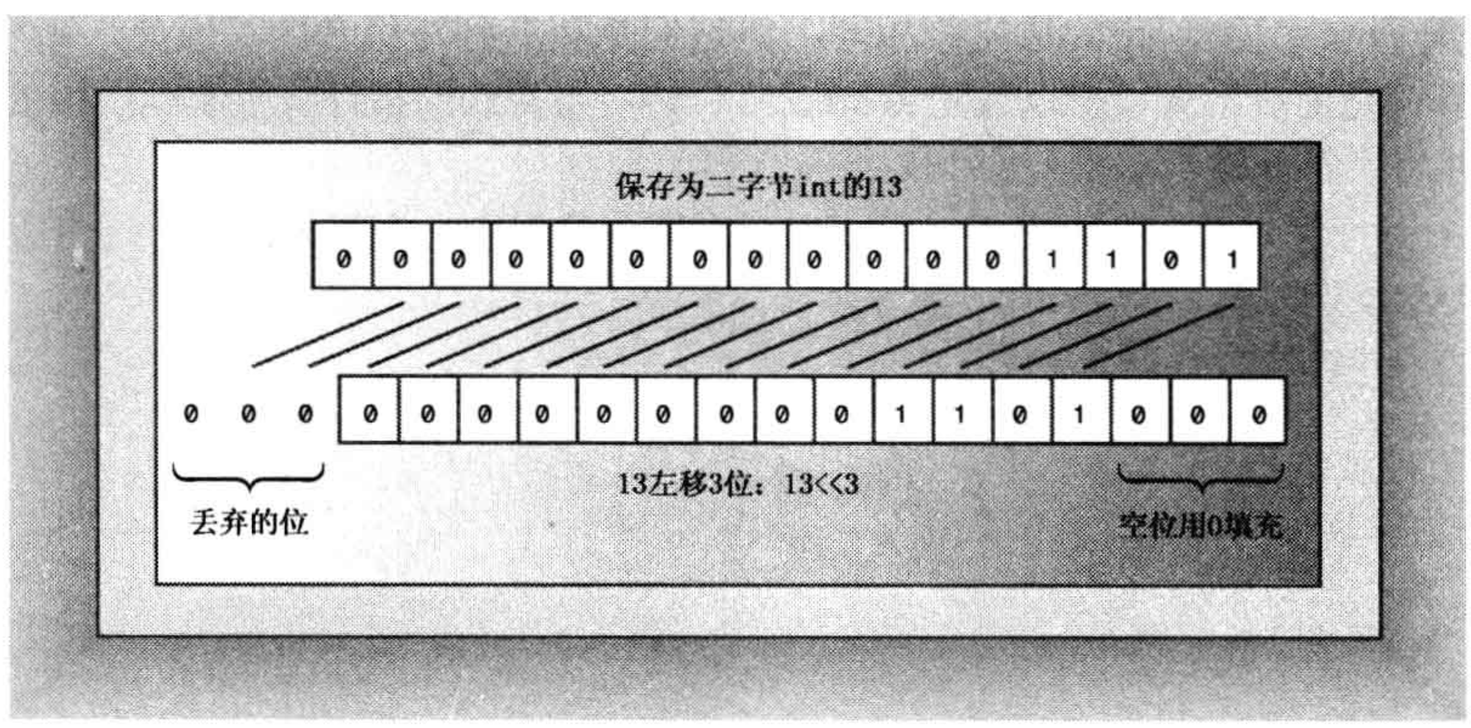

图 E.1　左移运算符

左移运算符提供了通常可以在汇编语言中找到的功能。不过，左移运算符在汇编语言中会直接修改寄存器的内容，而 C++左移运算符生成一个新值，而不修改原来的值。例如，请看下面的代码：

```
int x = 20;
int y = x << 3;
```

上述代码不会修改 x 的值。表达式 x<<3 使用 x 的值来生成一个新值，就像 x+3 会生成一个新值，而不会修改 x 一样。

如果要用左移运算符来修改变量的值，则还必须使用赋值运算符。可以使用常规的赋值运算符或<<=运算符（该运算符将移动与赋值结合在一起）：

```
x = x << 4;            // regular assignment
y <<= 2;               // shift and assign
```

正如所期望的，右移运算符（>>）将位向右移，其语法如下：

```
value >> shift
```

其中，value 是要移动的整数值，shift 是要移动的位数。例如，下面的代码将值 17 中所有的位向右移两位：

```
17 >> 2
```

对于无符号整数，腾出的位置用 0 填充，超过边界的位被删除。对于有符号整数，腾出的位置可能用 0 填充，也可能用原来最左边的位填充，这取决于 C++实现（图 E.2 是一个用 0 填充的例子）。

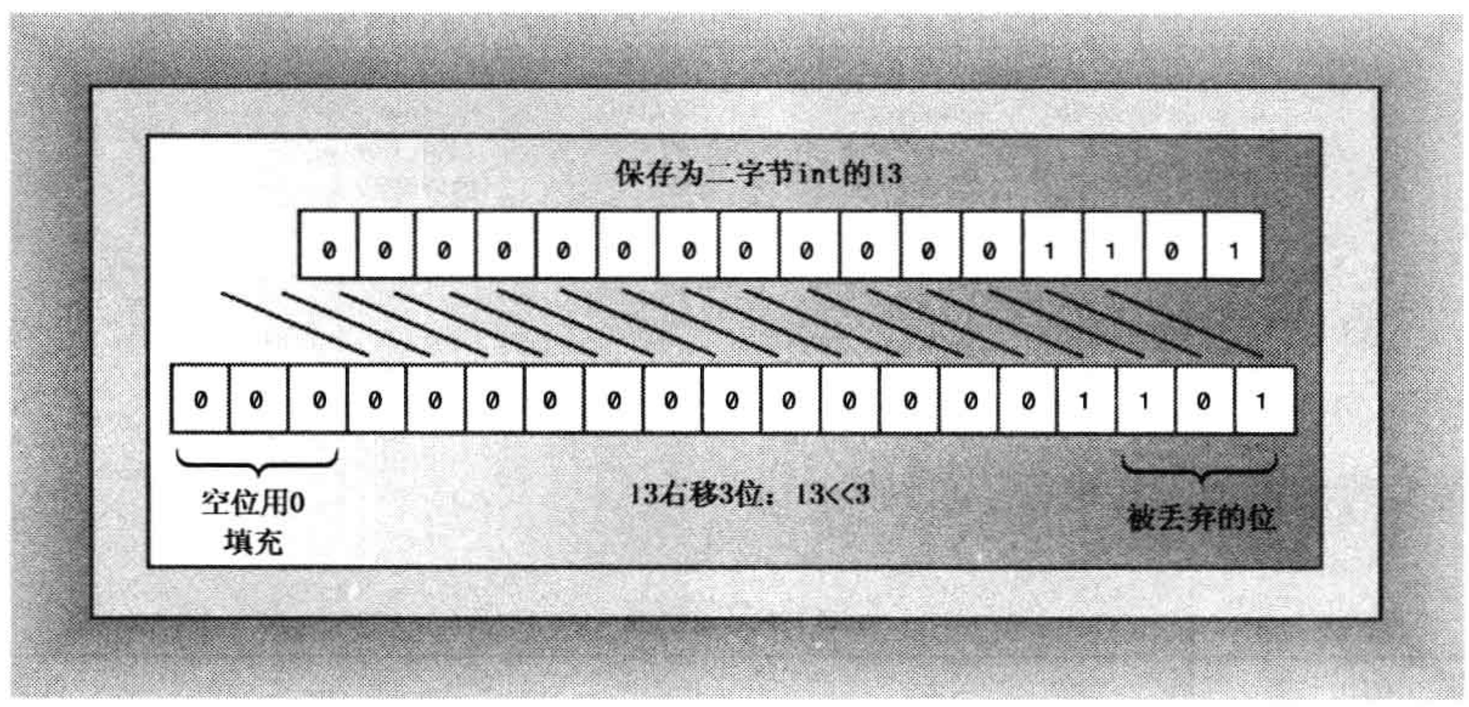

图 E.2　右移运算符

向右移动一位相当于除以 2。向右移动 n 位相当于除以 2^n。

C++还定义了一个“右移并赋值”运算符，如果要用移动后的值替换变量的值，可以这样做：

```
int q = 43;
q >>= 2;               // replace 43 by 43 >> 2, or 10
```

在有些系统上，使用左移运算符（右移运算符）实现将整数乘（除）以 2 的速度比使用乘（除）法运算符更快，但由于编译器在优化代码方面越来越好，因此这种差异正在减小。

E.1.2　逻辑按位运算符

逻辑按位运算符类似于常规的逻辑运算符，只是它们用于值的每一位，而不是整个值。例如，请看常规的非运算符（!）和位非（或求反）运算符（～）。!运算符将 true（或非零值）转换为 false，将 false 值转换为 true。～运算符将每一位转换为它的反面（1 转换为 0，0 转换为 1）。例如，对于 unsigned char 值 3：

```
unsigned char x = 3;
```

表达式!x 的值为 0。要知道~x 的值，先把它写成二进制形式：00000011。然后将每个 0 转换为 1，将每个 1 转换为 0。这样将得到值 11111100，在十进制中，为 252（图 E.3 是一个 16 位的例子）。新值是原值的补值。

按位运算符 OR（|）对两个整数值进行操作，生成一个新的整数值。如果被操作的两个值的对应位至少有一个为 1，则新值中相应位为 1，否则为 0（参见图 E.4）。

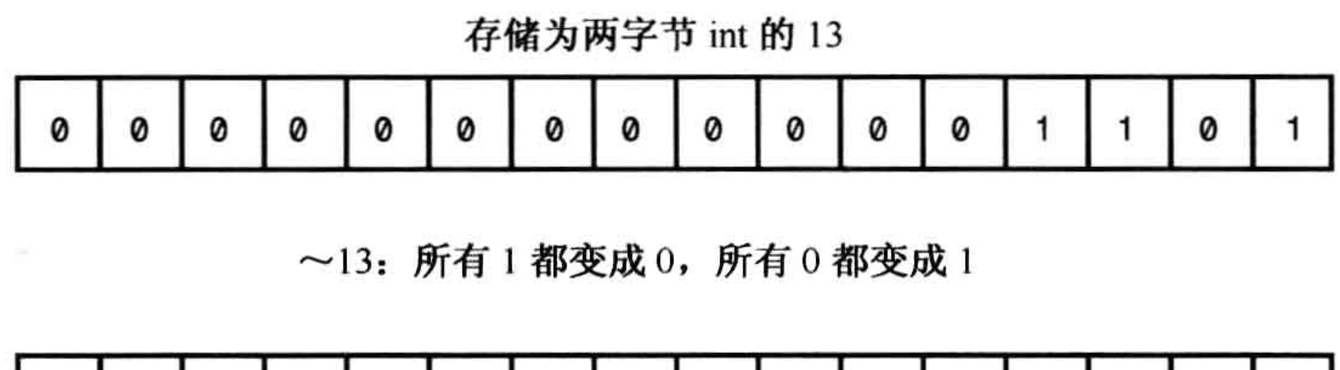

图 E.3　按位非运算符

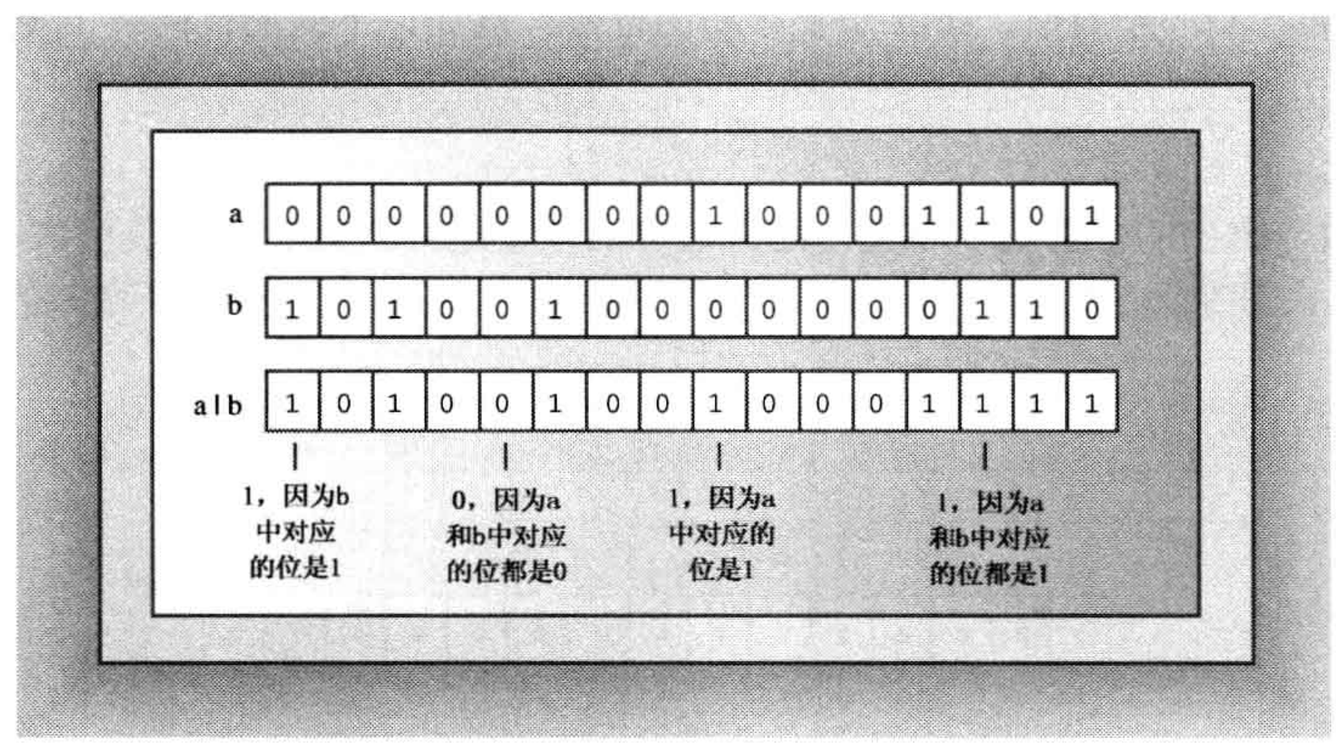

图 E.4　按位运算符 OR

表 E.1 对 | 运算符的操作方式进行了总结。

表 E.1　　b1|b2 的值

位　值	b1 = 0	b1 = 1
b2 = 0	0	1
b2 = 1	1	1

运算符| =组合了按位运算符 OR 与赋值运算符的功能：

```
a |= b;  // set a to a | b
```

按位运算符 XOR（^）将两个整数值结合起来，生成一个新的整数值。如果原始值中对应的位有一个（而不是两个）为 1，则新值中相应位为 1；如果对应的位都为 0 或 1，则新值中相应位为 0（参见图 E.5）。

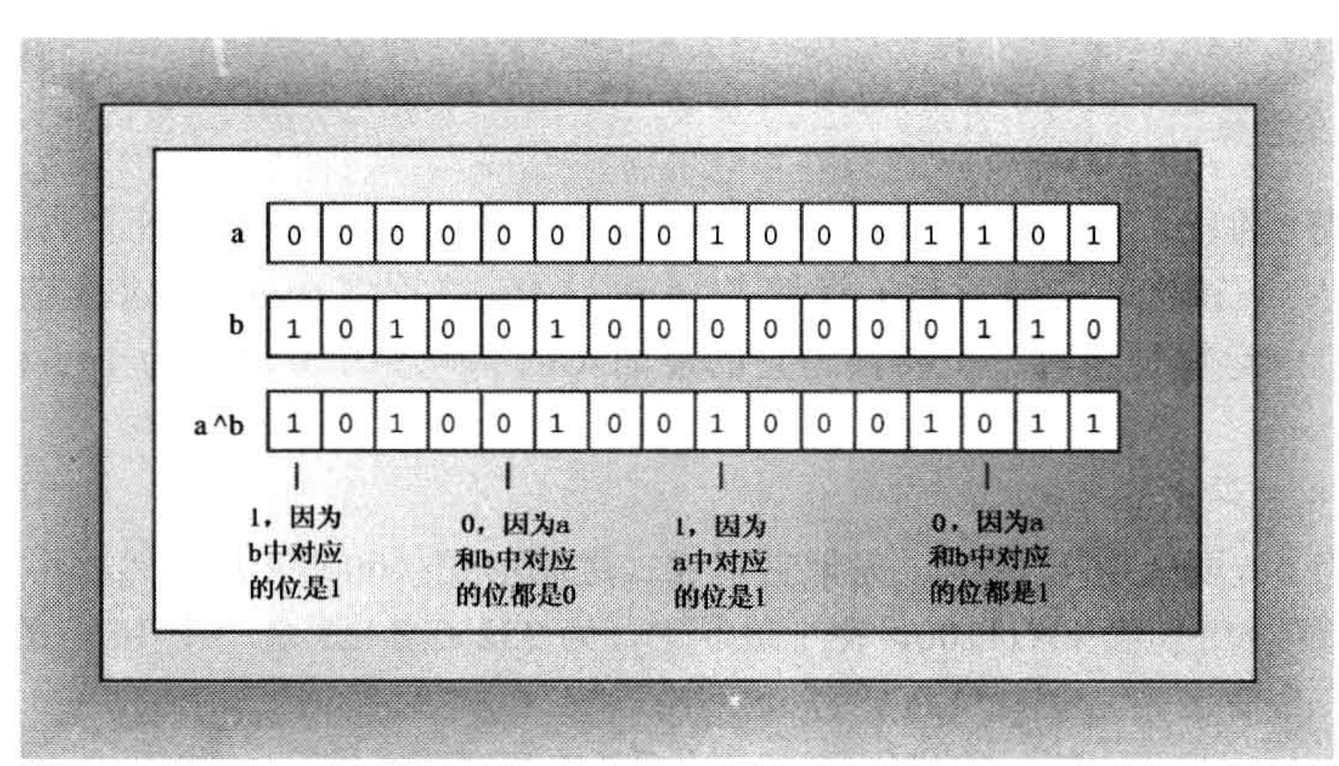

图 E.5　按位运算符 XOR

表 E.2 总结了^运算符的结合方式。

表 E.2　　b1^b2 的值

位　　值	b1 = 0	b1 = 1
b2 = 0	0	1
b2 = 1	1	0

^ =运算符结合了按位运算符 XOR 和赋值运算符的功能：

```
a ^= b;  // set a to a ^ b
```

按位运算符 AND（&）将两个整数结合起来，生成一个新的整数值。如果原始值中对应位都为 1，则新值中相应位为 1，否则为 0（参见图 E.6）。

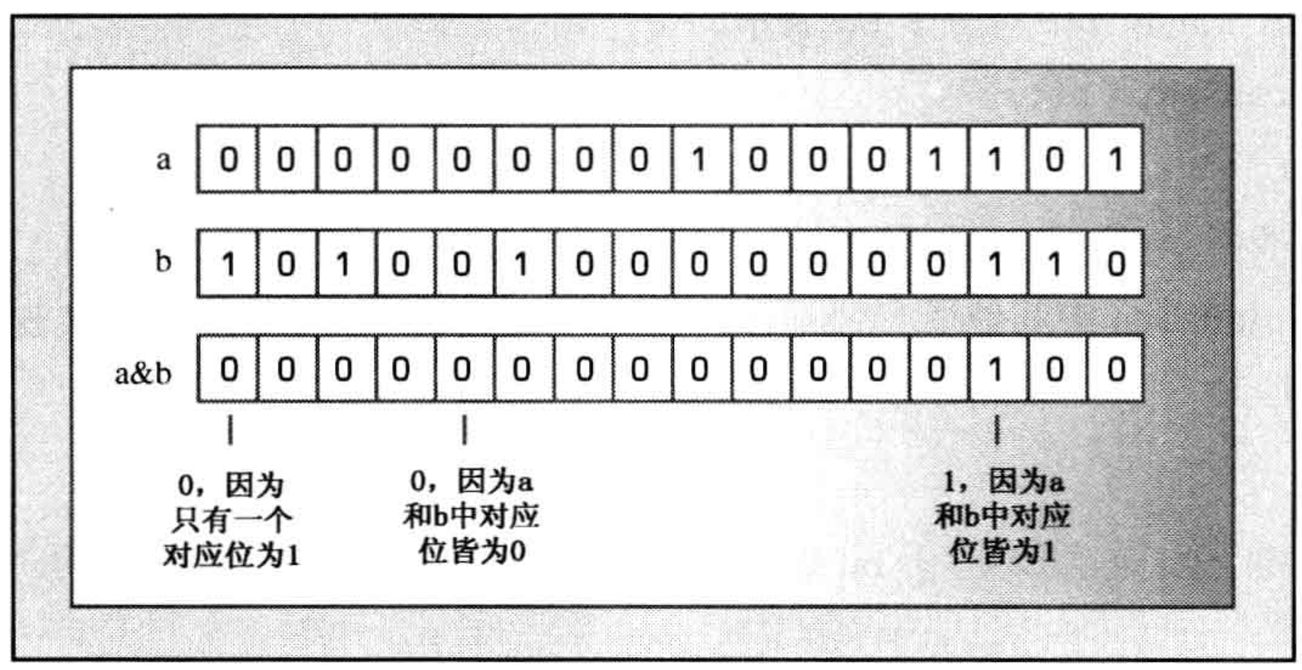

图 E.6　按位运算符 AND

表 E.3 总结了&运算符是如何运算的。

表 E.3　　b1&b2 的值

位　　值	b1 = 0	b2 = 1
b2 = 0	0	0
b2 = 1	0	1

& =运算符结合了按位运算符 AND 和赋值运算符的功能：

```
a &= b;  // set a to a & b
```

E.1.3　按位运算符的替代表示

对于几种按位运算符，C++提供了替代表示，如表 E.4 所示。它们适用于字符集中不包含传统按位运算符的区域。

表 E.4　　按位运算符的替代表示

标 准 表 示	替 代 表 示
&	bitand
& =	and_eq
|	bitor
|=	or_eq
~	compl
^	xor
^ =	xor_eq

这些替代表示让您能够编写下面这样的语句：

```
b = compl a bitand b;  // same as b = ~a & b;
c = a xor b;           // same as c = a ^ c;
```

E.1.4 几种常用的按位运算符技术

控制硬件时，常涉及打开/关闭特定的位或查看它们的状态。按位运算符提供了执行这种操作的途径。下面简要地介绍一下这些方法。

在下面的示例中，lottabits 表示一个值，bit 表示特定位的值。位从右到左进行编号，从 0 开始，因此，第 n 位的值为 2^n。例如，只有第 3 位为 1 的整数的值为 2^3（8）。一般来说，正如附录 A 介绍的，各个位都对应于 2 的幂。因此我们使用术语位（bit）表示 2 的幂；这对应于特定位为 1，其他所有位都为 0 的情况。

1. 打开位

下面两项操作打开 lottabits 中对应于 bit 表示的位：

```
lottabits = lottabits | bit;
lottabits |= bit;
```

它们都将对应的位设置为 1，而不管这一位以前的值是多少。这是因为对 1 和 1 或者 0 和 1 执行 OR 操作时，都将得到 1。lottabits 中其他所有位都保持不变，这是因为对 0 和 0 做 OR 操作将得到 0，对 1 和 0 做 OR 操作将生成 1。

2. 切换位

下面两项操作切换 lottabits 中对应于 bit 表示的位。也就是说，如果位是关闭的，则将被打开；如果位是打开的，将被关闭：

```
lottabits = lottabits ^ bit;
lottabits ^= bit;
```

对 0 和 1 执行 XOR 操作的结果为 1，因此将关闭已打开的位；对 1 和 1 执行 XOR 操作的结果为 0，因此将打开已关闭的位。lottabits 中其他所有位都保持不变，这是因为对 0 和 0 执行 XOR 操作的结果为 0，对 1 和 0 执行 XOR 操作的结果为 1。

3. 关闭位

下面的操作将关闭 lottabits 中对应于 bit 表示的位：

```
lottabits = lottabits & ~bit;
```

该语句关闭相应的位，而不管它以前的状态如何。首先，运算符～bit 将原来为 1 的位设置为 0，原来为 0 的位设置为 1。对 0 和任意值执行 AND 操作都将得到 0，因此关闭相应的位。lottabits 中其他所有位都保持不变，这是因为对 1 和任意值执行 AND 操作时，该位的值将保持不变。

下面是一种更简洁的方法：

```
lottabits &= ~bit;
```

4. 测试位的值

如果要确定 lottabits 中对应于 bit 的位是否为 1，则下面的测试不一定管用：

```
if (lottabits == bit)            // no good
```

这是因为即使 lottabits 中对应的位为 1，而其他位也可能为 1。仅当对应的位为 1，而其他位皆为 0 时，上述等式才为 true。因此修补的方式是，首先对 lottabits 和 bit 执行 AND 操作，这样生成的值的对应位保持不变，因为对 1 和任何值执行 AND 操作都将保持该值不变；而其他位都为 0，因为对 0 和任何值执行 AND 操作的结果都为 0。正确的测试如下：

```
if (lottabits & bit == bit)      // testing a bit
```

实际应用中，程序员常将上述测试简化为：

```
if (lottabits & bit)       // testing a bit
```

因为 bit 中有一位为 1，而其他位都为 0，因此 lottabits & bit 的结果要么为 0（测试结果为 false），要么为 bit（非零值，测试结果为 true）。

E.2　成员解除引用运算符

C++允许定义指向类成员的指针，对这种指针进行声明或解除引用时，需要使用一种特殊的表示法。为说明需要使用的特殊表示法，先来看一个样本类：

```
class Example
{
private:
    int feet;
    int inches;
public:
    Example();
    Example(int ft);
    ~Example();
    void show_in() const;
    void show_ft() const;
    void use_ptr() const;
};
```

如果没有具体的对象，则 inches 成员只是一个标签。也就是说，这个类将 inches 定义为一个成员标识符，但要为它分配内存，必须声明这个类的一个对象：

```
Example ob;  // now ob.inches exists
```

因此，可以结合使用标识符 inches 和特定的对象，来引用实际的内存单元（对于成员函数，可以省略对象名，但对象被认为是指针指向的对象）。

C++允许这样定义一个指向标识符 inches 的成员指针：

```
int Example::*pt = &Example::inches;
```

这种指针与常规指针有所差别。常规指针指向特定的内存单元，而 pt 指针并不指向特定的内存单元，因为声明中没有指出具体的对象。指针 pt 指的是 inches 成员在任意 Example 对象中的位置。和标识符 inches 一样，pt 被设计为与对象标识符一起使用。实际上，表达式*pt 对标识符 inches 的角色做了假设，因此，可以使用对象标识符来指定要访问的对象，使用 pt 指针来指定该对象的 inches 成员。例如，类方法可以使用下面的代码：

```
int Example::*pt = &Example::inches;
Example ob1;
Example ob2;
Example *pq = new Example;
cout << ob1.*pt << endl; // display inches member of ob1
cout << ob2.*pt << endl; // display inches member of ob2
cout << po->*pt << endl; // display inches member of *po
```

其中，*和->*都是成员解除引用运算符（member dereferencing operator）。声明对象（如 obl）后，ob1.*pi 指的将是 ob1 对象的 inches 成员。同样，pq->*pt 指的是 pq 指向的对象的 inches 成员。

改变上述示例中使用的对象，将改变使用的 inches 成员。不过也可以修改 pt 指针本身。由于 feet 的类型与 inches 相同，因此可以将 pt 重新设置为指向 feet 成员（而不指向 inches 成员），这样 ob1.*pt 将是 ob1 的 feet 成员：

```
pt = &Example::feet;       // reset pt
cout << ob1.*pt << endl;   // display feet member of ob1
```

实际上，*pt 相当于一个成员名，因此可用于标识（相同类型的）其他成员。

也可以使用成员指针来标识成员函数，其语法稍微复杂点。对于不带任何参数、返回值为 void 的函数，声明一个指向该函数的指针的代码如下：

```
void (*pf)();  // pf points to a function
```

声明指向成员函数的指针时，必须指出该函数所属的类。例如，下面的代码声明了一个指向 Example 类方法的指针：

```
void (Example::*pf)() const;  // pf points to an Example member function
```

这表明 pf 可用于可使用 Example 方法的地方。注意，Example::*pf 必须放在括号中，可以将特定成员函数的地址赋给该指针：

```
pf = &Example::show_inches;
```

注意，和普通函数指针的赋值情况不同，这里必须使用地址运算符。完成赋值操作后，便可以使用一个对象来调用该成员函数：

```
Example ob3(20);
(ob3.*pf)();     // invoke show_inches() using the ob3 object
```

必须将 ob3.*pf 放在括号中，以明确地指出，该表达式表示的是一个函数名。

由于 show_feet( )的原型与 show_inches( )相同，因此也可以使用 pf 来访问 show_feet( )方法：

```
pf = &Example::show_feet;
(ob3.*pf)();     // apply show_feet() to the ob3 object
```

程序清单 E.1 中的类定义包含一个 use_ptr( )方法，该方法使用成员指针来访问 Example 类的数据成员和函数成员。

程序清单 E.1　memb_pt.cpp

```
// memb_pt.cpp -- dereferencing pointers to class members
#include <iostream>
using namespace std;

class Example
{
private:
    int feet;
    int inches;
public:
    Example();
    Example(int ft);
    ~Example();
    void show_in() const;
    void show_ft() const;
    void use_ptr() const;
};

Example::Example()
{
    feet = 0;
    inches = 0;
}

Example::Example(int ft)
{
    feet = ft;
```

```
    inches = 12 * feet;
}

Example::~Example()
{
}
void Example::show_in() const
{
    cout << inches << " inches\n";
}

void Example::show_ft() const
{
    cout << feet << " feet\n";
}

void Example::use_ptr() const
{
    Example yard(3);
    int Example::*pt;
    pt = &Example::inches;
    cout << "Set pt to &Example::inches:\n";
    cout << "this->pt: " << this->*pt << endl;
    cout << "yard.*pt: " << yard.*pt << endl;
    pt = &Example::feet;
    cout << "Set pt to &Example::feet:\n";
    cout << "this->pt: " << this->*pt << endl;
    cout << "yard.*pt: " << yard.*pt << endl;
    void (Example::*pf)() const;
    pf = &Example::show_in;
    cout << "Set pf to &Example::show_in:\n";
    cout << "Using (this->*pf)(): ";
    (this->*pf)();
    cout << "Using (yard.*pf)(): ";
    (yard.*pf)();
}

int main()
{
    Example car(15);
    Example van(20);
    Example garage;

    cout << "car.use_ptr() output:\n";
    car.use_ptr();
    cout << "\nvan.use_ptr() output:\n";
    van.use_ptr();

    return 0;
}
```

下面是程序清单 E.1 中程序的运行情况：

```
car.use_ptr() output:
Set pt to &Example::inches:
this->pt: 180
```

```
yard.*pt: 36
Set pt to &Example::feet:
this->pt: 15
yard.*pt: 3
Set pf to &Example::show_in:
Using (this->*pf)(): 180 inches
Using (yard.*pf)(): 36 inches

van.use_ptr() output:
Set pt to &Example::inches:
this->pt: 240
yard.*pt: 36
Set pt to &Example::feet:
this->pt: 20
yard.*pt: 3
Set pf to &Example::show_in:
Using (this->*pf)(): 240 inches
Using (yard.*pf)(): 36 inches
```

这个例子在编译期间给指针赋值。在更复杂的类中，可以使用指向数据成员和方法的成员指针，以便在运行阶段确定与指针关联的成员。

E.3 alignof（C++11）

计算机系统可能限制数据在内存中的存储方式。例如，一个系统可能要求 double 值存储在编号为偶数的内存单元中，而另一个系统可能要求其起始地址为 8 个整数倍。运算符 alignof 将类型作为参数，并返回一个整数，指出要求的对齐方式。例如，对齐要求可能决定结构中信息的组织方式，如程序清单 E.2 所示。

程序清单 E.2 align.cpp

```
// align.cpp -- checking alignment
#include <iostream>
using namespace std;
struct things1
{
    char ch;
    int a;
    double x;
};
struct things2
{

    int a;
    double x;
    char ch;
};

int main()
{
    things1 th1;
    things2 th2;
    cout << "char alignment: " << alignof(char) << endl;
```

```
    cout << "int alignment: " << alignof(int) << endl;
    cout << "double alignment: " << alignof(double) << endl;
    cout << "things1 alignment: " << alignof(things1) << endl;
    cout << "things2 alignment: " << alignof(things2) << endl;
    cout << "things1 size: " << sizeof(things1) << endl;
    cout << "things2 size: " << sizeof(things2) << endl;
    return 0;
}
```

下面是该程序在一个系统中的输出：

```
char alignment: 1
int alignment: 4
double alignment: 8
things1 alignment: 8
things2 alignment: 8
things1 size: 16
things2 size: 24
```

两个结构的对齐要求都是 8。这意味着结构长度将是 8 的整数倍，这样创建结构数组时，每个元素的起始位置都是 8 的整数倍。在程序清单 E.2 中，每个结构的所有成员只占用 13 位，但结构要求占用的位数为 8 的整数倍，这意味着需要填充一些位。在每个结构中，double 成员的对齐要求为 8 的整数倍，但在结构 thing1 和 thing2 中，成员的排列顺序不同，这导致 thing2 需要更多的内部填充，以便其边界处于正确的位置。

E.4　noexcept（C++11）

关键字 noexcept 用于指出函数不会引发异常。它也可用作运算符，判断操作数（表达式）是否可能引发异常；如果操作数可能引发异常，则返回 false，否则返回 true。例如，请看下面的声明：

```
int hilt(int);
int halt(int) noexcept;
```

表达式 noexcept(hilt) 的结果为 false，因为 hilt( ) 的声明未保证不会引发异常，但 noexcept(halt) 的结果为 true。

附录 F　模板类 string

本附录的技术性较强，但如果您只想了解模板类 string 的功能，可以将重点放在对各种 string 类方法的描述上。

string 类是基于下述模板定义的：

```
template<class charT, class traits = char_traits<charT>,
                 class Allocator = allocator<charT> >
class basic_string {...};
```

其中，chatT 是存储在字符串中的类型；traits 参数是一个类，它定义了类型要被表示为字符串时，所必须具备的特征。例如，它必须有 length( )方法，该方法返回被表示为 charT 数组的字符串的长度。这种数组结尾用 charT(0)值（广义的空值字符）表示。（表达式 charT(0)将 0 转换为 charT 类型。它可以像类型为 char 时那样为零，也可以是 charT 的一个构造函数创建的对象）。这个类还包含用于对值进行比较等操作的方法。Allocator 参数是用于处理字符串内存分配的类。默认的 allocator<char>模板按标准方式使用 new 和 delete。

有 4 种预定义的具体化：

```
typedef basic_string<char> string;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
typedef basic_string<wchar_t> wstring;
```

上述具体化又使用下面的具体化：

```
char_traits<char>
allocator<char>
char_traits<char16_t>
allocator<char_16>
char_traits<char_32>
allocator<char_32>
char_traits<wchar_t>
allocator<wchar_t>
```

除 char 和 wchar_t 外，还可以通过定义 traits 类和使用 basic_string 模板来为其他一些类型创建一个 string 类。

F.1　13 种类型和一个常量

模板 basic_string 定义了几种类型，供以后定义方法时使用：

```
typedef traits                                  traits_type;
typedef typename traits::char_type              value_type;
typedef Allocator                               allocator_type;
typedef typename Allocator::size_type           size_type;
typedef typename Allocator::difference_type     difference_type;
typedef typename Allocator::reference           reference;
typedef typename Allocator::const_reference     const_reference;
typedef typename Allocator::pointer             pointer;
typedef typename Allocator::const_pointer       const_pointer;
```

traits 是对应于特定类型（如 char_traits<char>）的模板参数；traits_type 将成为该特定类型的 typedef。下述表示法意味着 char_type 是 traits 表示的类中定义的一个类型名：

```
typedef typename traits::char_type          value_type;
```

关键字 typename 告诉编译器，表达式 trait::char_type 是一种类型。例如，对于 string 具体化，value_type 为 char。

size_type 与 size_of 的用法相似，只是它根据存储的类型返回字符串的长度。对于 string 具体化，将根据 char 返回字符串的长度，在这种情况下，size_type 与 size_of 等效。size_type 是一种无符号类型。

differrence_type 用于度量字符串中两个元素之间的距离（单位为元素的长度）。通常，它是底层类型 size_type 有符号版本。

对于 char 具体化来说，pointer 的类型为 char *，而 reference 的类型为 char &类型。然而，如果要为自己设计的类型创建具体化，则这些类型（pointer 和 reference）可以指向类，与基本指针和引用有相同的特征。

为将标准模板库（STL）算法用于字符串，该模板定义了一些迭代器类型：

```
typedef (models random access iterator)          iterator;
typedef (models random access iterator)          const_iterator;
typedef std::reverse_iterator<iterator>          reverse_iterator;
typedef std::reverse_iterator<const_iterator>    const_reverse_iterator;
```

该模板还定义了一个静态常量：

```
static const size_type npos = -1;
```

由于 size_type 是无符号的，因此将-1 赋给 npos 相当于将最大的无符号值赋给它，这个值比可能的最大数组索引大 1。

F.2　数据信息、构造函数及其他

可以根据其效果来描述构造函数。由于类的私有部分可能依赖于实现，因此可根据公用接口中可用的数据来描述这些效果。表 F.1 列出了一些方法，它们的返回值可用来描述构造函数和其他方法的效果。注意，其中的大部分术语来自 STL。

表 F.1　一些 string 数据方法

方　法	返　回　值
begin( )	指向字符串第一个字符的迭代器
cbegin( )	一个 const_iterator，指向字符串中的第一个字符（C++11）
end( )	为超尾值的迭代器
cend( )	为超尾值的 const_iterator（C++11）
rbegin( )	为超尾值的反转迭代器
crbegin( )	为超尾值的反转 const_iterator（C++11）
rend( )	指向第一个字符的反转迭代器
crend( )	指向第一个字符的反转 const_iterator（C++11）
size( )	字符串中的元素数，等于 begin( )到 end( )之间的距离
length( )	与 size( )相同
capacity( )	给字符串分配的元素数。这可能大于实际的字符数，capacity( ) – size( )的值表示在字符串末尾附加多少字符后需要分配更多的内存
max_size( )	字符串的最大长度

续表

方　法	返　回　值
data( )	一个指向数组第一个元素的 const charT*指针，其第一个 size( )元素等于*this 控制的字符串中对应的元素，其下一个元素为 charT 类型的 charT(0)字符（字符串末尾标记）。当 string 对象本身被修改后，该指针可能无效
c_str( )	一个指向数组第一个元素的 const charT*指针，其第一个 size( )元素等于*this 控制的字符串中对应的元素，其下一个元素是 charT 类型的 charT（0）字符（字符串尾标识）。当 string 对象本身被修改后，该指针可能无效
get_allocator( )	用于为字符串 object 分配内存的 allocator 对象的副本

请注意 begin( )、rend( )、data( )和 c_str( )之间的差别。它们都与字符串的第一个字符相关，但相关的方式不同。begin( )和 rend( )方法返回一个迭代器，正如第 16 章讨论的，这是一种广义指针。具体地说，begin( )返回一个正向迭代器模型，而 rend( )返回反转迭代器的一个副本。这两种方法都引用了 string 对象管理的字符串（由于 string 类使用动态内存分配，因此实际的 string 内容不一定位于对象中，因此，我们使用术语“管理”来描述对象和字符串之间的关系）。可以将返回迭代器的方法用于基于迭代器的 STL 算法中。例如，可以使用 STL reverse( )函数来反转字符串的内容：

```
string word;
cin >> word;
reverse(word.begin(), word.end());
```

而 data( )和 c_str( )方法返回常规指针。另外，返回的指针将指向存储字符串字符的数组的第一个元素。该数组可能（但不一定）是 string 对象管理的字符串的副本（string 对象采用的内部表示可以是数组，但不一定非得是数组）。由于返回的指针可能指向原始数据，而原始数据是 const，因此不能用它们来修改数据。另外，当字符串被修改后，将不能保证这些指针是有效的，这表明它们可能指向原始数据。data( )和 c_str( )的区别在于，c_str( )指向的数组以空值字符（或与之等价的其他字符）结束，而 data( )只是确保实际的字符串字符是存在的。因此，c_str( )方法期望接受一个 C-风格字符串参数：

```
string file("tofu.man");
ofstream outFile(file.c_str());
```

同样，data( )和 size( )可用作这种函数的参数，即接受指向数组元素的指针和表示要处理的元素数目的值：

```
string vampire("Do not stake me, oh my darling!");
int vlad = byte_check(vampire.data(), vampire.size());
```

C++实现可能将 string 对象的字符串表示为动态分配的 C-风格字符串，并使用 char*指针来实现正向迭代器。在这种情况下，实现可能让 begin( )、data( )和 c_str( )都返回同样的指针，但返回指向 3 个不同的数据对象的引用也是合法的（虽然更复杂）。

在 C++11 中，模板类 basic_string 有 11 个构造函数（在 C++98 中只有 6 个）和一个析构函数：

```
explicit basic_string(const Allocator& a = Allocator());
basic_string(const charT* s, const Allocator& a = Allocator());
basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
basic_string(const basic_string& str);
basic_string(const basic_string& str, const Allocator&);
basic_string(const basic_string& str, size_type pos,
    size_type n = npos, const Allocator& a = Allocator());
basic_string(basic_string&& str) noexcept;
basic_string(const basic_string&& str, const Allocator&);
basic_string(size_type n, charT c, const Allocator& a = Allocator());
template<class InputIterator>
basic_string(InputIterator begin, InputIterator end,
```

```
    const Allocator& a = Allocator());
basic_string(initializer_list<charT>, const Allocator& = Allocator());
~basic_string();
```

有些新增的构造函数以不同的方式处理参数。例如，C++98 包含如下复制构造函数：

```
basic_string(const basic_string& str, size_type pos = 0,
    size_type n = npos, const Allocator& a = Allocator());
```

而 C++11 用三个构造函数取代了它——上述列表中的第 2～4 个，这提高了编码效率。真正新增的只有移动构造函数（使用右值引用的构造函数，这在第 18 章讨论过）以及使用 initializer_list 参数的构造函数。

注意到大多数构造函数构造函数都有一个下面这样的参数：

```
const Allocator& a = Allocator()
```

Allocator 是用于管理内存的 allocator 类的模板参数名；Allocator( )是这个类的默认构造函数。因此，在默认情况下，构造函数将使用 allocator 对象的默认版本，但它们使得能够选择使用 allocator 对象的其他版本。下面分别介绍这些构造函数。

F.2.1　默认构造函数

默认构造函数的原型如下：

```
explicit basic_string(const Allocator& a = Allocator());
```

通常，接受 allocator 类的默认参数，并使用该构造函数来创建空字符串：

```
string bean;
wstring theory;
```

调用该默认构造函数后，将存在下面的关系：

- data( )方法返回一个非空指针，可以将该指针加上 0；
- size( )方法返回 0；
- capacity( )的返回值不确定。

将 data( )返回的值赋给指针 str 后，第一个条件意味着 str + 0 是有效的。

F.2.2　使用 C-风格字符串的构造函数

使用 C-风格字符串的构造函数让您能够将 string 对象初始化为一个 C-风格字符串；从更普遍的意义上看，它使得能够将 charT 具体化初始化为一个 charT 数组：

```
basic_string(const charT* s, const Allocator& a = Allocator());
```

为确定要复制的字符数，该构造函数将 traits::length( )方法用于 s 指向的数组（s 不能为空指针）。例如，下面的语句使用指定的字符串来初始化 toast 对象：

```
string toast("Here's looking at you, kid.");
```

char 类型的 traits::length( )方法将使用空值字符来确定要复制多少个字符。

该构造函数被调用后，将存在下面的关系：

- data( )方法返回一个指针，该指针指向数组 s 的一个副本的第一个元素；
- size( )方法返回的值等于 trains::length( )的值；
- capacity( )方法返回一个至少等于 size( )的值。

F.2.3　使用部分 C-风格字符串的构造函数

使用部分 C-风格字符串的构造函数让您能够使用 C-风格字符串的一部分来初始化 string 对象；从更广泛的意义上说，该构造函数使得能够使用 charT 数组的一部分来初始化 charT 具体化：

```
basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
```

该构造函数将 s 指向的数组中的 n 个字符复制到构造的对象中。请注意，如果 s 包含的字符数少于 n，则复

制过程将不会停止。如果 n 大于 s 的长度，该构造函数将把字符串后面的内存内容解释为 charT 类型的数据。

该构造函数要求 s 不能是空值指针，同时 n<npos（npos 是一个静态类常量，它是字符串可能包含的最大元素数目）。如果 n 等于 npos，该构造函数将引发一个 out_of_range 异常（由于 n 的类型为 size_type，而 npos 是 size_type 的最大值，因此 n 不能大于 npos）；否则，在该构造函数被调用后，将存在下面的关系：

- data( )方法返回一个指针，该指针指向数组 s 的副本的第一个元素；
- size( )方法返回 n；
- capacity( )方法返回一个至少等于 size( )的值。

F.2.4 使用左值引用的构造函数

复制构造函数类似于下面这样：

```
basic_string(const basic_string& str);
```

它使用一个 string 参数初始化一个新的 string 对象：

```
string mel("I'm ok!");
string ida(mel);
```

其中，ida 将是 mel 管理的字符串副本。

下一个构造函数要求您指定一个分配器：

```
basic_string(const basic_string& str, const Allocator&);
```

调用这两个构造函数中的任何一个后，将存在如下关系：

- data( )方法返回一个指针，该指针指向分配的数组副本，该数组的第一个元素是 str.data( )指向的；
- size( )方法返回 str.size()的值；
- capacity( )方法返回一个至少等于 size( )的值。

再下一个构造函数让您能够指定多项内容：

```
basic_string(const basic_string& str, size_type pos, size_type n = npos,
    const Allocator& a = Allocator());
```

第二个参数（pos）指定了源字符串中的位置，将从这个位置开始进行复制：

```
string att("Telephone home.");
string et(att, 4);
```

位置编号从 0 开始，因此，位置 4 是字符 p。所以，et 被初始化为“phone home”。

第 3 个参数 n 是可选的，它指定要复制的最大字符数目，因此下面的语句将 pt 初始化为字符串“phone”：

```
string att("Telephone home.");
string pt(att, 4, 5);
```

然而，该构造函数不能跨越源字符串的结尾，例如，下面的语句将在复制句点后停止：

```
string pt(att, 4, 200)
```

因此，该构造函数实际复制的字符数量等于 n 和 str.size( )-pos 中较小的一个。

该构造函数要求 pos 不大于 str.size( )，也就是说，被复制的初始位置必须位于源字符串中。如果情况并非如此，该构造函数将引发 out_of_range 异常；否则，该构造函数被调用后，copy_len 将是 n 和 str.size( )-pos 中较小的一个，并存在下面的关系：

- data( )方法返回一个指向字符串的指针，该字符串包含 copy_len 个元素，这些元素是从 str 的 pos 位置开始复制而得到的；
- size( )方法返回 copy_len；
- capacity( )方法返回一个不小于 size( )的值。

F.2.5 使用右值引用的构造函数（C++11）

C++11 给 string 类添加了移动语义。正如第 18 章介绍的，这意味着添加一个移动构造函数，它使用右

值引用而不是左值引用：

```
basic_string(basic_string&& str) noexcept;
```

在实参为临时对象时将调用这个构造函数：

```
string one("din");      // C-style string constructor
string two(one);        // copy constructor - one is an lvalue
string three(one+two); // move constructor, sum is an rvalue
```

正如第 18 章讨论的，three 将获取 operator + ()创建的对象的所有权，而不是将该对象复制给 three，再销毁原始对象。

第二个使用右值引用的构造函数让您能够指定分配器：

```
basic_string(const basic_string&& str, const Allocator&);
```

调用这两个构造函数中的任何一个后，将存在如下关系：

- data( )方法返回一个指针，该指针指向分配的数组副本，该数组的第一个元素是 str.data( )指向的；
- size( )方法返回 str.size()的值；
- capacity( )方法返回一个至少等于 size( )的值。

F.2.6　使用一个字符的 n 个副本的构造函数

使用一个字符的 n 个副本的构造函数创建一个由 n 个 c 组成的 string 对象：

```
basic_string(size_type n, charT c, const Allocator& a = Allocator());
```

该构造函数要求 n<npos。如果 n 等于 npos，该构造函数将引发 out_of_range 异常；否则，该构造函数被调用后，将存在下面的关系：

- data( )方法返回一个指向字符串第一个元素的指针，该字符串由 n 个元素组成，其中每个元素的值都为 c；
- size( )方法返回 n；
- capacity( )方法返回不小于 size( )的值。

F.2.7　使用区间的构造函数

使用区间的构造函数使用一个用迭代器定义的、STL-风格的区间：

```
template<class InputIterator>
basic_string(InputIterator begin, InputIterator end,
                const Allocator& a = Allocator());
```

begin 迭代器指向源字符串中要复制的第一个元素，end 指向要复制的最后一个元素的后面。这种构造函数可用于数组、字符串或 STL 容器：

```
char cole[40] = "Old King Cole was a merry old soul.";
string title(cole + 4, cole + 8);
vector<char> input;
char ch;
while (cin.get(ch) && ch != '\n')
    input.push_back(ch);
string str_input(input.begin(), input.end());
```

在第一种用法中，InputIterator 的类型为 const char *；在第二种用法中，InputIterator 的类型为 vector<char>::iterator。

调用该构造函数后，将存在下面的关系：

- data( )方法返回一个指向字符串的第一个元素的指针，该字符串是通过复制区间[begin，end）中的元素得到的；
- size( )方法返回 begin 到 end 之间的距离（度量距离时，使用的单位为对迭代器解除引用得到的数

据类型的长度）；

- capacity( )方法返回一个不小于 size( )的值。

F.2.8 使用初始化列表的构造函数（C++11）

这个构造函数接受一个 initializer_list<charT>参数：

```
basic_string(initializer_list<charT> il, const Allocator& a = Allocator());
```

可将一个用大括号括起的字符列表作为参数：

```
string slow({'s', 'n', 'a', 'i', 'l'});
```

这并非初始化 string 的最方便方式，但让 string 的接口类似于 STL 容器类。

initializer_list 类包含成员函数 begin( )和 end( )，调用该构造函数的影响与调用使用区间的构造函数相同：

```
basic_string(il.begin(), il.end(), a);
```

F.2.9 内存杂记

有些方法用于处理内存，如清除内存的内容、调整字符串长度或容量。表 F.2 列出了一些与内存相关的方法。

表 F.2 一些与内存有关的方法

方　法	作　用
void resize(size_type n)	如果 n>npos，将引发 out_of_range 异常；否则，将字符串的长度改为 n，如果 n<size( )，则截短字符串，如果 n>size( )，则使用 charT(0)中的字符填充字符串
void resize(size_type n, charT c)	如果 n>npos，将引发 out_of_range 异常；否则，将字符串长度改为 n，如果 n<size( )，则截短字符串，如果 n>size( )，则使用字符 c 填充字符串
void reserve(size_type res_arg = 0)	将 capacity( )设置为大于或等于 res_arg。由于这将重新分配字符串，因此以前的引用、迭代器和指针将无效
void shrink_to_fit( )	请求让 capacity( )的值与 size( )相同，这是 C++11 新增的
void clear( ) noexcept	删除字符串中所有的字符
bool empty( )const noexcept	如果 size( )==0，则返回 true

F.3 字符串存取

有 4 种方法可以访问各个字符，其中两种方法使用[ ]运算符，另外两种方法使用 at( )方法：

```
reference operator[](size_type pos);
const_reference operator[](size_type pos) const;
reference at(size_type n);
const_reference at(size_type n) const;
```

第一个 operator[ ]( )方法使得能够使用数组表示法来访问字符串的元素，可用于检索或更改值。第二个 operator[ ]( )方法可用于 const 对象，但只能用于检索值：

```
string word("tack");
cout << word[0];    // display the t
word[3] = 't';      // overwrite the k with a t
const ward("garlic");
cout << ward[2];    // display the r
```

at( )方法提供了相似的访问功能，只是索引是通过函数参数提供的：

```
string word("tack");
cout << word.at(0);    // display the t
```

差别在于（除语法差别外）：at( )方法执行边界检查，如果 pos>=size( )，将引发 out_of_range 异常。pos 的类型为 size_type，是无符号的，因此 pos 的值不能为负；而 operator[ ]( )方法不进行边界检查，因此，如果 pos>=size( )，则其行为将是不确定的（如果 pos= =size( )，const 版本将返回空值字符的等价物）。

因此，可以在安全性（使用 at( )检测异常）和执行速度（使用数组表示）之间进行选择。

还有一个这样的函数，它返回原始字符串的子字符串：

```
basic_string substr(size_type pos = 0, size_type n = npos) const;
```

它返回一个字符串——这是从 pos 开始，复制 n 个字符（或到字符串尾部）得到的。例如，下面的代码将 pet 初始化为“donkey”：

```
string message("Maybe the donkey will learn to sing.");
string pet(message.substr(10, 6));
```

C++11 新增了如下四个存取方法：

```
const charT& front() const;
charT& front();
const charT& back() const;
charT& back();
```

其中 front( )方法访问 string 的第一个元素，相当于 operator[] (0)；back( )方法访问 string 的最后一个元素，相当于 operator[] (size( ) - 1)。

F.4　基本赋值

在 C++11 中，有 5 个重载的赋值方法，在 C++98 的基础上增加了两个：

```
basic_string& operator=(const basic_string& str);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);
basic_string& operator=(basic_string&& str) noexcept;  // C++11
basic_string& operator=(initializer_list<charT>);      // C++11
```

第一个方法将一个 string 对象赋给另一个；第二个方法将 C-风格字符串赋给 string 对象；第三个方法将一个字符赋给 string 对象；第四个方法使用移动语义，将一个右值 string 对象赋给一个 string 对象；第五个方法让您能够使用初始化列表进行赋值。因此，下面的操作都是可能的：

```
string name("George Wash");
string pres, veep, source, join, awkward;
pres = name;
veep = "Road Runner";
source = 'X';
join = name + source;   // now with move semantics!
awkward = {'C','l','o','u','s','e','a','u'};
```

F.5　字符串搜索

string 类提供了 6 种搜索函数，其中每个函数都有 4 个原型。下面简要地介绍它们。

F.5.1 find( )系列

在 C++11 中，find( )的原型如下：

```
size_type find (const basic_string& str, size_type pos = 0) const noexcept;
size_type find (const charT* s, size_type pos = 0) const;
size_type find (const charT* s, size_type pos, size_type n) const;
size_type find (charT c, size_type pos = 0) const noexcept;
```

第一个返回 str 在调用对象中第一次出现时的起始位置。搜索从 pos 开始，如果没有找到子字符串，将返回 npos。

下面的代码在一个字符串中查找字符串“hat”的位置：

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.find(shorter);            // sets loc1 to 1
size_type loc2 = longer.find(shorter, loc1 + 1); // sets loc2 to 16
```

由于第二条搜索语句从位置 2 开始（That 中的 a），因此它找到的第一个 hat 位于字符串尾部。要测试是否失败，可使用 string::npos 值：

```
if (loc1 == string::npos)
    cout << "Not found\n";
```

第二个方法完成同样的工作，但它使用字符数组而不是 string 对象作为子字符串：

```
size_type loc3 = longer.find("is");               //sets loc3 to 5
```

第三个方法完成相同的工作，但它只使用字符串 s 的前 n 个字符。这与使用 basic_string（const charT * s，size_type n）构造函数，然后将得到的对象用作第一种格式的 find( )的 string 参数的效果完全相同。例如，下面的代码搜索子字符串“fun”：

```
size_type loc4 = longer.find("funds", 3);         //sets loc4 to 10
```

第四个方法的功能与第一个相同，但它使用一个字符而不是 string 对象作为子字符串：

```
size_type loc5 = longer.find('a');                //sets loc5 to 2
```

F.5.2 rfind( )系列

rfind( )方法的原型如下：

```
size_type rfind(const basic_string& str,
                size_type pos = npos) const noexcept;
size_type rfind(const charT* s, size_type pos = npos) const;
size_type rfind(const charT* s, size_type pos, size_type n) const;
size_type rfind(charT c, size_type pos = npos) const noexcept;
```

这些方法与相应 find( )方法的工作方式相似，但它们搜索字符串最后一次出现的位置，该位置位于 pos 之前（包括 pos）。如果没有找到，该方法将返回 npos。

下面的代码从字符串末尾开始查找子字符串“hat”的位置：

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.rfind(shorter);           // sets loc1 to 16
size_type loc2 = longer.rfind(shorter, loc1 - 1); // sets loc2 to 1
```

F.5.3 find_first_of( )系列

find_first_of( )方法的原型如下：

```
size_type find_first_of(const basic_string& str,
                        size_type pos = 0) const noexcept;
```

```
size_type find_first_of(const charT* s, size_type pos, size_type n) const;
size_type find_first_of(const charT* s, size_type pos = 0) const;
size_type find_first_of(charT c, size_type pos = 0) const noexcept;
```

这些方法与对应 find( )方法的工作方式相似，但它们不是搜索整个子字符串，而是搜索子字符串中的字符首次出现的位置。

```
string longer("That is a funny hat.");
string shorter("fluke");
size_type loc1 = longer.find_first_of(shorter);  // sets loc1 to 10
size_type loc2 = longer.find_first_of("fat");    // sets loc2 to 2
```

在 longer 中，首次出现的 fluke 中的字符是 funny 中的 f，而首次出现的 fat 中的字符是 That 中的 a。

F.5.4　find_last_of( )系列

find_last_of( )方法的原型如下：

```
size_type find_last_of (const basic_string& str,
                        size_type pos = npos) const noexcept;
size_type find_last_of (const charT* s, size_type pos, size_type n) const;
size_type find_last_of (const charT* s, size_type pos = npos) const;
size_type find_last_of (charT c, size_type pos = npos) const noexcept;
```

这些方法与对应 rfind( )方法的工作方式相似，但它们不是搜索整个子字符串，而是搜索子字符串中的字符出现的最后位置。

下面的代码在一个字符串中查找字符串“hat”和“any”中字母最后出现的位置：

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.find_last_of(shorter);  // sets loc1 to 18
size_type loc2 = longer.find_last_of("any");    // sets loc2 to 17
```

在 longer 中，最后出现的 hat 中的字符是 hat 中的 t，而最后出现的 any 中的字符是 hat 中的 a。

F.5.5　find_first_not_of( )系列

find_first_not_of( )方法的原型如下：

```
size_type find_first_not_of(const basic_string& str,
                            size_type pos = 0) const noexcept;
size_type find_first_not_of(const charT* s, size_type pos,
                            size_type n) const;
size_type find_first_not_of(const charT* s, size_type pos = 0) const;
size_type find_first_not_of(charT c, size_type pos = 0) const noexcept;
```

这些方法与对应 find_first_of( )方法的工作方式相似，但它们搜索第一个不位于子字符串中的字符。下面的代码在字符串中查找第一个没有出现在“This”和“Thatch”中的字母：

```
string longer("That is a funny hat.");
string shorter("This");
size_type loc1 = longer.find_first_not_of(shorter);  // sets loc1 to 2
size_type loc2 = longer.find_first_not_of("Thatch"); // sets loc2 to 4
```

在 longer 中，That 中的 a 是第一个在 This 中没有出现的字符，而字符串 longer 中的第一个空格是第一个没有在 Thatch 中出现的字符。

F.5.6　find_last_not_of( )系列

find_last_not_of( )方法的原型如下：

```
size_type find_last_not_of (const basic_string& str,
                            size_type pos = npos) const noexcept;
size_type find_last_not_of (const charT* s, size_type pos,
                            size_type n) const;
size_type find_last_not_of (const charT* s,
                            size_type pos = npos) const;
size_type find_last_not_of (charT c, size_type pos = npos) const noexcept;
```

这些方法与对应 find_last_of( )方法的工作方式相似，但它们搜索的是最后一个没有在子字符串中出现的字符。

下面的代码在字符串中查找最后一个没有出现在“That.”中的字符：

```
string longer("That is a funny hat.");
string shorter("That.");
size_type loc1 = longer.find_last_not_of(shorter);     // sets loc1 to 15
size_type loc2 = longer.find_last_not_of(shorter, 10); // sets loc2 to 10
```

在 longer 中，最后的空格是最后一个没有出现在 shorter 中的字符，而 longer 字符串中的 f 是搜索到位置 10 时，最后一个没有出现在 shorter 中的字符。

F.6 比较方法和函数

string 类提供了用于比较 2 个字符串的方法和函数。下面是方法的原型：

```
int compare(const basic_string& str) const noexcept;

int compare(size_type pos1, size_type n1,
            const basic_string& str) const;
int compare(size_type pos1, size_type n1,
            const basic_string& str,
            size_type pos2, size_type n2) const;
int compare(const charT* s) const;
int compare(size_type pos1, size_type n1, const charT* s) const;
int compare(size_type pos1, size_type n1,
            const charT* s, size_type n2 ) const;
```

这些方法使用 traits::compare( )方法，后者是为用于字符串的字符类型定义的。如果根据 traits::compare( )提供的顺序，第一个字符串位于第二个字符串之前，则第一个方法将返回一个小于 0 的值；如果这两个字符串相同，则它将返回 0；如果第一个字符串位于第二个字符串的后面，则它将返回一个大于 0 的值。如果较长的字符串的前半部分与较短的字符串相同，则较短的字符串将位于较长的字符串之前。

```
string s1("bellflower");
string s2("bell");
string s3("cat");
int a13 = s1.compare(s3); // a13 is < 0
int a12 = s1.compare(s2); // a12 is > 0
```

第二个方法与第一个方法相似，但它进行比较时，只使用第一个字符串中从位置 pos1 开始的 n1 个字符。

下面的示例将字符串 s1 的前 4 个字符同字符串 s2 进行比较：

```
string s1("bellflower");
string s2("bell");
int a2 = s1.compare(0, 4, s2); // a2 is 0
```

第三个方法与第一个方法相似，但它使用第一个字符串中从 pos1 位置开始的 n1 个字符和第二个字符串

中从 pos2 位置开始的 n2 个字符进行比较。例如，下面的语句将对 stout 中的 out 和 about 中的 out 进行比较：

```
string st1("stout boar");
string st2("mad about ewe");
int a3 = st1.compare(2, 3, st2, 6, 3);  // a3 is 0
```

第四个方法与第一个方法相似，但它将一个字符数组而不是 string 对象作为第二个字符串。

第五和六个方法与第三个方法相似，但将一个字符串数组而不是 string 对象作为第二个字符串。

非成员比较函数是重载的关系运算符：

```
operator==()
operator<()
operator<=()
operator>()
operator>=()
operator!=()
```

每一个运算符都被重载，使之将 string 对象与 string 对象进行比较、将 string 对象与 C-风格字符串进行比较、将 C-风格字符串与 string 对象进行比较。它们都是根据 compare( )方法定义的，因此提供了一种在表示方面更为方便的比较方式。

F.7　字符串修改方法

string 类提供了多个用于修改字符串的方法，其中绝大多数都拥有大量的重载版本，因此可用于 string 对象、字符串数组、单个字符和迭代器区间。

F.7.1　用于追加和相加的方法

可以使用重载的+ =运算符或 append( )方法将一个字符串追加到另一个字符串的后面。如果得到的字符串长于最大字符串长度，将引发 length_error 异常。+=运算符使得能够将 string 对象、字符串数组或单个字符追加到 string 对象的后面：

```
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
```

append( )方法也使得能够将 string 对象、字符串数组或单个字符追加到 string 对象的后面。此外，通过指定初始位置和追加的字符数，或者通过指定区间，还可以追加 string 对象的一部分。通过指定要使用字符串中的多少个字符，可以追加字符串的一部分。追加字符的版本使得能够指定要复制该字符的多少个实例。下面是各种 append( )方法的原型：

```
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos,
                     size_type n);
template<class InputIterator>
  basic_string& append(InputIterator first, InputIterator last);
basic_string& append(const charT* s);
basic_string& append(const charT* s, size_type n);
basic_string& append(size_type n, charT c);  // append n copies of c
void push_back(charT c);                     // appends 1 copy of c
```

下面是几个示例：

```
string test("The");
test.append("ory");  // test is "Theory"
test.append(3,'!');  // test is "Theory!!!"
```

operator+( )函数被重载，以便能够拼接字符串。该重载函数不修改字符串，而是创建一个新的字符串，该字符串是通过将第二个字符串追加到第一个字符串后面得到的。加法函数不是成员函数，它们使得能够将 string 对象和 string 对象、string 对象和字符串数组、字符串数组和 string 对象、string 对象和字符以及字符和 string 对象相加。下面是一些例子：

```
string st1("red");
string st2("rain");
string st3 = st1 + "uce";  // st3 is "reduce"
string st4 = 't' + st2;    // st4 is "train"
string st5 = st1 + st2;    // st5 is "redrain"
```

F.7.2 其他赋值方法

除了基本的赋值运算符外，string 类还提供了 assign( )方法，该方法使得能够将整个字符串、字符串的一部分或由相同字符组成的字符序列赋给 string 对象。下面是各种 assign( )方法的原型：

```
basic_string& assign(const basic_string& str);
basic string& assign(basic_string&& str) noexcept;   // C++11
basic_string& assign(const basic_string& str, size_type pos,
                     size_type n);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c); // assign n copies of c
template<class InputIterator>
  basic_string& assign(InputIterator first, InputIterator last);
basic_string& assign(initializer_list<charT>);  // C++11
```

下面是几个例子：

```
string test;
string stuff("set tubs clones ducks");
test.assign(stuff, 1, 5);   // test is "et tu"
test.assign(6, '#");        // test is "######"
```

接受右值引用作为参数的 assign( )方法是 C++11 新增的，它支持移动语义；另一个新增的 assign( )方法让您能够将 initializer_list 赋给 string 对象。

F.7.3 插入方法

insert( )方法使得能够将 string 对象、字符串数组或几个字符插入到 string 对象中。这个方法与 append( )方法相似，但它还接受另一个指定插入位置的参数，该参数可以是位置，也可以是迭代器。数据将被插入到插入点的前面。有几种方法返回一个指向得到的字符串的引用。如果 pos1 超过了目标字符串结尾，或者 pos2 超过了要插入的字符串结尾，该方法将引发 out_of_range 异常。如果得到的字符串长于最大长度，该方法将引发 length_error 异常。下面是各种 insert( )方法的原型：

```
basic_string& insert(size_type pos1, const basic_string& str);
basic_string& insert(size_type pos1, const basic_string& str,
                     size_type pos2, size_type n);
basic_string& insert(size_type pos, const charT* s, size_type n);
basic_string& insert(size_type pos, const charT* s);
basic_string& insert(size_type pos, size_type n, charT c);
iterator insert(const_iterator p, charT c);
iterator insert(const_iterator p, size_type n, charT c);
template<class InputIterator>
  void insert(iterator p, InputIterator first, InputIterator last);
iterator insert(const_iterator p, initializer_list<charT>); // C++11
```

例如，下面的代码将字符串“former”字符串插入到“The banker.”中 b 的前面：

```
string st3("The banker.");
st3.insert(4, "former ");
```

而下面的代码将字符串“ waltzed”（不包括!，它是第 9 个字符）插入到“The former banker.”末尾的句号之前：

```
st3.insert(st3.size() - 1, " waltzed!", 8);
```

F.7.4　清除方法

erase( )方法从字符串中删除字符，其原型如下：

```
basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(const_iterator position);
iterator erase(const_iterator first, iterator last);
void pop_back();
```

第一种格式将从 pos 位置开始，删除 n 个字符或删除到字符串尾。第二种格式删除迭代器位置引用的字符，并返回指向下一个元素的迭代器；如果后面没有其他元素，则返回 end( )。第三种格式删除区间[first，last）中的字符，即从 first（包括）到 last（不包括）之间的字符；它返回最后一个迭代器，该迭代器指向最后一个被删除的元素后面的一个元素。最后，方法 pop_back( )删除字符串中的最后一个字符。

F.7.5　替换方法

各种 replace( )方法都指定了要替换的字符串部分和用于替换的内容。可以使用初始位置和字符数目或迭代器区间来指定要替换的部分。替换内容可以是 string 对象、字符串数组，也可以是特定字符的多个实例。对于用于替换的 string 对象和数组，可以通过指定特定部分（使用位置和计数或只使用计数）或迭代器区间做进一步的修改。下面是各种 replace( )方法的原型：

```
basic_string& replace(size_type pos1, size_type n1, const basic_string& str);
basic_string& replace(size_type pos1, size_type n1, const basic_string& str,
                      size_type pos2, size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s,
                      size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s);
basic_string& replace(size_type pos, size_type n1, size_type n2, charT c);
basic_string& replace(const_iterator i1, const_iterator i2,
                      const basic_string& str);
basic_string& replace(const_iterator i1, const_iterator i2,
                      const charT* s, size_type n);
basic_string& replace(const_iterator i1, const_iterator i2,
                      const charT* s);
basic_string& replace(const_iterator i1, const_iterator i2,
                      size_type n, charT c);
template<class InputIterator>
  basic_string& replace(const_iterator i1, const_iterator i2,
                        InputIterator j1, InputIterator j2);
basic_string& replace(const_iteraor i1, const_iteator i2,
                      initializer)list<charT> il);
```

下面是一个例子：

```
string test("Take a right turn at Main Street.");
test.replace(7,5,"left"); // replace right with left
```

注意，您可以使用 find( )来找出要在 replace( )中使用的位置：

```
string s1 = "old";
string s2 = "mature";
string s3 = "The old man and the sea";
string::size_type pos = s3.find(s1);
if (pos != string::npos)
    s3.replace(pos, s1.size(), s2);
```

上述代码将 old 替换为 mature。

F.7.6 其他修改方法：copy( )和 swap( )

copy( )方法将 string 对象或其中的一部分复制到指定的字符串数组中：

```
size_type copy(charT* s, size_type n, size_type pos = 0) const;
```

其中，s 指向目标数组，n 是要复制的字符数，pos 指出从 string 对象的什么位置开始复制。复制将一直进行下去，直到复制了 n 个字符或到达 string 对象的最后一个字符。函数返回复制的字符数，该方法不追加空值字符，同时由程序员负责检查数组的长度是否足够存储复制的内容。

警告：copy( )*方法不追加空值字符，也不检查目标数组的长度是否足够。*

swap( )方法使用一个时间恒定的算法来交换两个 string 对象的内容：

```
void swap(basic_string& str);
```

F.8 输出和输入

string 类重载了<<运算符来显示 string 对象，该运算符返回 istream 对象的引用，因此可以拼接输出：

```
string claim("The string class has many features.");
cout << claim << endl;
```

string 类重载了>>运算符，使得能够将输入读入到字符串中：

```
string who;
cin >> who;
```

到达文件尾、读取字符串允许的最大字符数或遇到空白字符后，输入将终止（空白的定义取决于字符集以及 charT 表示的类型）。

有两个 getline( )函数，第一个的原型如下：

```
template<class charT, class traits, class Allocator>
  basic_istream<charT,traits>& getline(basic_istream<charT,traits>& is,
              basic_string<charT,traits,Allocator>& str, charT delim);
```

这个函数将输入流 is 中的字符读入到字符串 str 中，直到遇到定界字符 delim、到达文件尾或者到达字符串的最大长度。delim 字符将被读取（从输入流中删除），但不被存储。第二个版本没有第三个参数，同时使用换行符（或其广义形式），而不是 delim：

```
string str1, str2;
getline(cin, str1);       // read to end-of-line
getline(cin, str2, '.');  // read to period
```

附录 G　标准模板库方法和函数

标准模板库（STL）旨在提供通用算法的高效实现，它通过通用函数（可用于满足特定算法要求的任何容器）和方法（可用于特定容器类实例）来表达这些算法。本附录假设您对 STL 有一定的了解（如通过阅读第 16 章）。例如，本章假设您了解迭代器和构造函数。

G.1　STL 和 C++11

C++11 对 C++语言做了大量修改，本书无法全面介绍，同样 C++11 对 STL 也做了大量修改，本附录无法全面介绍。然而，可以对新增的内容做一总结。

C++11 给 STL 新增了多个元素。首先，它新增了多个容器；其次，给旧容器新增了多项功能；第三，在算法系列中新增了一些模板函数。本附录介绍了所有这些变化，对前两类变化有大致了解将很有帮助。

G.1.1　新增的容器

C++11 新增了如下容器：array、forward_list、unordered_st 以及无序关联容器 unordered_multiset、unordered_map 和 unordered_multimap。

array 容器一旦声明，其长度就是固定的，它使用静态（栈）内存，而不是动态分配的内存。提供它旨在替代数组；array 受到的限制比 vector 多，但效率更高。

容器 list 是一种双向链表，除两端的节点外，每个节点都链接到它前面和后面的节点。forward_list 是一种单向链表，除最后一个节点外，每个节点都链接到下一个节点。相对于 list，它更紧凑，但受到的限制更多。

与 set 和其他关联容器一样，无序关联容器让您能够使用键快速检索数据，差别在于关联容器使用的底层数据结构为树，而无序关联容器使用的是哈希表。

G.1.2　对 C++98 容器所做的修改

C++11 对容器类的方法做了三项主要修改。

首先，新增的右值引用使得能够给容器提供移动语义（参见第 18 章）。因此，STL 现在给容器提供了移动构造函数和移动赋值运算符，这些方法将右值引用作为参数。

其次，由于新增了模板类 initilizer_list（参见第 18 章），因此新增了将 initilizer_list 作为参数的构造函数和赋值运算符。这使得可以编写类似于下面的代码：

```
vector<int> vi{100, 99, 97, 98};
vi = {96, 99, 94, 95, 102};
```

第三，新增的可变参数模板（variadic template）和函数参数包（parameter pack）使得可以提供就地创建（emplacement）方法。这意味着什么呢？与移动语义一样，就地创建旨在提高效率。请看下面的代码段：

```
class Items
{
    double x;
    double y;
    int m;
```

```
public:
    Items();                                // #1
    Items (double xx, double yy, int mm); // #2
    ...
};
...
vector<Items> vt(10);
...
vt.push_back(Items(8.2, 2.8, 3));  //
```

调用 insert( )将导致内存分配函数在 vt 末尾创建一个默认 Items 对象。接下来，构造函数 Items( )创建一个临时 Items 对象，该对象被复制到 vt 的开头，然后被删除。在 C++11 中，您可以这样做：

```
vi.emplace_back(8.2, 2.8, 3);
```

方法 emplace_back( )是一个可变参数模板，将一个函数参数包作为参数：

```
template <class... Args> void emplace_back(Args&&... args);
```

上述三个实参（8.2、2.8 和 3）将被封装到参数 args 中。参数 args 被传递给内存分配函数，而内存分配函数将其展开，并使用接受三个参数的 Items 构造函数（#2），而不是默认构造函数（#1）。也就是说，它使用 Items(args...)，这里将展开为 Items(8.2, 2.8, 3)。因此，将在矢量中就地创建所需的对象，而不是创建一个临时对象，再将其复制到矢量中。

STL 在多个就地创建方法中使用了这种技术。

G.2 大部分容器都有的成员

所有容器都定义了表 G.1 列出的类型。在这个表中，x 为容器类型，如 vector<int>；T 为存储在容器中的类型，如 int。表 G.1 中的示例阐明了含义

表 G.1　　为所有容器定义的类型

类　型	值
x::value-type	T，元素类型
x::reference	T &
x::const_reference	const T &
x::iterator	指向 T 的迭代器类型，行为与 T*相似
x::const_iterator	指向 const T 的迭代器类型，行为与 const T *相似
x::different_type	用于表示两个迭代器之间距离的符号整型，如两个指针的差
x::size_type	无符号整型 size_type 可以表示数据对象的长度、元素数目和下标

类定义使用 typedef 定义这些成员。可以使用这些类型来声明适当的变量。例如，下面的代码使用迂回的方式，将由 string 对象组成的矢量中的第一个“bonus”替换为“bogus”，以演示如何使用成员类型来声明变量。

```
using namespace std;
vector<string> input;
string temp;
while (cin >> temp && temp != "quit")
    input.push_back(temp);
vector<string>::iterator want=
```

```
    find(input.begin(), input.end(), string("bonus"));
if (want != input.end())
{
    vector<string>::reference r = *want;
    r = "bogus";
}
```

上述代码使 r 成为一个指向（want 指向的）input 中元素的引用。同样，继续前面的例子，可以编写下面这样的代码：

```
vector<string>::value_type s1 = input[0];  // s1 is type string
vector<string>::reference s2 = input[1];   // s2 is type string &
```

这将导致 s1 为一个新 string 对象，它是 input[0]的拷贝；而 s2 为指向 input[1]的引用。在这个例子中，由于已经知道模板是基于 string 类型的，因此编写下面的等效代码将更简单：

```
string s1 = input[0];      // s1 is type string
string & s2 = input[1];    // s2 is type string &
```

然而，还可以在更通用的代码中使用表 G.1 中较精致（其中容器和元素的类型是通用的）的类型。例如，假设希望 min( )函数将一个指向容器的引用作为参数，并返回容器中最小的项目。这假设为用于实例化模板的值类型定义了<运算符，而不想使用 STL min_element( )算法，这种算法使用迭代器接口。由于参数可能是 vector<int>、list<strint>或 deque<double>，因此需要使用带模板参数（如 Bag）的模板来表示容器（也就是说，Bag 是一个模板类型，可能被实例化为 vector<int>、list<string>或其他一些容器类型）。因此，函数的参数类型应为 const Bag & b。返回类型是什么呢？应为容器的值类型，即 Bag::value_type。然而，在这种情况下，Bag 只是一个模板参数，编译器无法知道 value_type 成员实际上是一种类型。但可以使用 typename 关键字来指出，类成员是 typedef：

```
vector<string>::value_type st;     // vector<string> a defined class
typename Bag::value_type m;        // Bag an as yet undefined type
```

对于上述第一个定义，编译器能够访问 vector 模板定义，该定义指出，value_type 是一个 typedef；对于第二个定义，typename 关键字指出，无论 Bag 将会是什么，Bag::value-type 都将是类型的名称。这些考虑因素导致了下面的定义：

```
template<typename Bag>
typename Bag::value_type min(const Bag & b)
{
    typename Bag::const_iterator it;
    typename Bag::value_type m = *b.begin();
    for (it = b.begin(); it != b.end(); ++it)
        if (*it < m)
            m = *it;
    return m;
}
```

这样，便可以这样使用该模板函数：

```
vector<int> temperatures;
// input temperature values into the vector
int coldest = min(temperatures);
```

temperatures 参数将使得 Bag 被谓词为 vector<int>，而 typename Bag::value-type 被谓词为 vector<int>::value_type，进而为 int。

所有的容器都还可以包含表 G.2 列出的成员函数或操作。其中，X 是容器类型，如 vector<int>；而 T 是存储在容器中的类型，如 int。另外，a 和 b 是类型为 X 的值；u 是标识符；r 是类型为 X 的非 const 值；rv 是类型为 X 的非 const 右值，而移动操作是 C++11 新增的。

表 G.2　为所有容器定义的操作

操　作	描　述
X u;	创建一个名为 u 的空对象
X( )	创建一个空对象
X(a)	创建对象 x 的拷贝
X u(a)	u 是 a 的拷贝（复制构造函数）
X u = a;	u 是 a 的拷贝（复制构造函数）
r = a	r 等于 a 的值（复制赋值）
X u(rv)	u 等于 rv 的原始值（移动构造函数）
X u = rv	u 等于 rv 的原始值（移动构造函数）
a = rv	u 等于 rv 的原始值（移动赋值）
(&a)->~X( )	对 a 的每个元素执行析构函数
begin( )	返回一个指向第一个元素的迭代器
end( )	返回一个指向超尾的迭代器
cbegin( )	返回一个指向第一个元素的 const 迭代器
cend( )	返回一个指向超尾的 const 迭代器
size( )	返回元素数目
maxsize( )	返回容器的最大可能长度
empty( )	如果容器为空，则返回 true
swap( )	交换两个容器的内容
= =	如果两个容器的长度相同、包含的元素相同且元素排列的顺序相同，则返回 true
! =	a!=b 返回!(a= =b)

使用双向或随机迭代器的容器（vector、list、deque、array、set 和 map）是可反转的，它们提供了表 G.3 所示的方法。

表 G.3　为可反转容器定义的类型和操作

操　作	描　述
X::reverse_iterator	指向类型 T 的反向迭代器
X::const_reverse_iterator	指向类型 T 的 const 反向迭代器
a.rbegin( )	返回一个反向迭代器，指向 a 的超尾
a.rend( )	返回一个指向 a 的开头的反向迭代器
a.crbegin( )	返回一个 const 反向迭代器，指向 a 的超尾
a.crend( )	返回一个指向 a 的开头的 const 反向迭代器

无序集合（set）和无序映射（map）无需支持表 G.4 所示的可选容器操作，但其他容器必须支持。

表 G.4　可选的容器操作

操　作	描　述
<	如果 a 按词典顺序排在 b 之前，则 a<b 返回 true
>	a>b 返回 b<a
<=	a<=b 返回!(a>b)
>=	a>=b 返回!(a<b)

容器的>运算符假设已经为值类型定义了>运算符。词典比较是一种广义的按字母顺序排序，它逐元素

地比较两个容器，直到两个容器中对应的元素相同时为止。在这种情况下，元素对的顺序将决定容器的顺序。例如，如果两个容器的前 10 个元素都相同，但第一个容器的第 11 个元素比第二个容器的第 11 个元素小，则第一个容器将排在第二个容器之前。如果两个容器中的元素一直相同，直到其中一个容器中的元素用完，则较短的容器将排在较长的容器之前。

G.3 序列容器的其他成员

模板类 vector、forward_list、list、deque 和 array 都是序列容器，它们都前面列出的方法，但 forward_list 不是可反转的，不支持表 G.3 所示的方法。序列容器以线性顺序存储一组类型相同的值。如果序列包含的元素数是固定的，通常选择使用 array；否则，应首先考虑使用 vector，它让 array 的随机存取功能以及添加和删除元素的功能于一身。然而，如果经常需要在序列中间添加元素，应考虑使用 list 或 forward_list。如果添加和删除操作主要是在序列两端进行的，应考虑使用 deque。

array 对象的长度是固定的，因此无法使用众多序列方法。表 G.5 列出除 array 外的序列容器可用的其他方法（forward_list 的 resize( )方法的定义稍有不同）。同样，其中 X 是容器类型，如 vector<int>；T 是存储在容器中的类型，如 int；a 是类型为 X 的值；t 是类型为 x::value_type 的左值或 const 右值；i 和 j 是输入迭代器；[i, j]是有效的区间；il 是类型为 initilizer_list<value_type>的对象；p 是指向 a 的有效 const 迭代器；q 是可解除引用的有效 const 迭代器；[q1, q2]是有效的 const 迭代器区间；n 是 x::size_type 类型的整数；Args 是模板参数包，而 args 是形式为 Args&&的函数参数包。

表 G.5　　为序列容器定义的其他操作

操　作	描　述
X(n, t)	创建一个序列容器，它包含 t 的 n 个拷贝
X a(n, t)	创建一个名为 a 的序列容器，它包含 t 的 n 个拷贝
X(i, j)	使用区间[i, j] 内的值创建一个序列容器
X a(i, j)	使用区间[i, j) 内的值创建一个名为 a 的序列容器
X(il)	创建一个序列容器，并将其初始化为 il 的内容
a = il;	将 il 的值复制到 a 中
a.emplace(p, args);	在 p 前面插入一个类型为 T 的对象，创建该对象时使用与 args 封装的参数匹配的构造函数
a.insert(p, t)	在 p 之前插入 t 的拷贝，并返回指向该拷贝的迭代器。T 的默认值为 T( )，即在没有显式初始化时，用于 T 类型的值
a.insert(p, rv)	在 p 之前插入 rv 的拷贝，并返回指向该拷贝的迭代器；可能使用移动语义
a.insert(p, n, t)	在 p 之前插入 t 的 n 个拷贝
a.insert(p, i, j)	在 p 之前插入[i, j)区间内元素的拷贝
a.insert(p, il)	等价于 a.insert(p, il.begin( ), il.end( ))
a.resize(n)	如果 n > a.size( )，则在 a.end( )之前插入 n - a.size( )个元素；用于新元素的值为没有显式初始化时，用于 T 类型的值；如果 n < a.size( )，则删除第 n 个元素之后的所有元素
a.resize(n, t)	如果 n > a.size( )，则在 a.end( )之前插入 t 的 n - a.size( )个拷贝；如果 n<a.size( )，则删除第 n 个元素之后的所有元素
a.assign(i, j)	使用区间[i，j)内的元素拷贝替换 a 当前的内容
a.assign(n, t)	使用 t 的 n 个拷贝替换 a 的当前内容。t 的默认值为 T( )，即在没有显式初始化时，用于 T 类型的值
a.assign(il)	等价于 a.assign(il.begin( ), il.end( ))
a.erase(q)	删除 q 指向的元素；返回一个指向 q 后面的元素的迭代器
a.erase(q1, q2)	删除区间[q1, q2]内的元素；返回一个迭代器，该迭代器指向 q2 原来指向的元素
a.clear( )	与 erase(a.begin( ), a.end( )) 等效
a.front( )	返回*a.begin( )（第一个元素）

表 G.6 列出了一些序列类（vector、forward_list、list 和 deque）都有的方法。

表 G.6 为某些序列定义的操作

操　作	描　述	容　器
a.back( )	返回*a.end( )（最后一个元素）	vector、list、deque
a.push_back(t)	将 t 插入到 a.end( )前面	vector、list、deque
a.push_back(rv)	将 rv 插入到 a.end( )前面；可能使用移动语义	vector、list、deque
a.pop_back( )	删除最后一个元素	vector、list、deque
a.emplace_back(args)	追加一个类型为 T 的对象，创建该对象时使用与 args 封装的参数匹配的构造函数	vector、list、deque
a.push_front(t)	将 t 的拷贝插入到第一个元素前面	forward_list、list、deque
a.push_front(rv)	将 rv 的拷贝插入到第一个元素前面；可能使用移动语义	forward_list、list、deque
a.emplace_front( )	在最前面插入一个类型为 T 的对象，创建该对象时使用与 args 封装的参数匹配的构造函数	forward_list、list、deque
a.pop_front( )	删除第一个元素	forward_list、list
a[n]	返回*(a.begin( )+ n)	vector、deque、array
a.at(n)	返回*(a.begin( )+ n)；如果 n>a.size，则引发 out_of_range 异常	vector、deque、array

模板 vector 还包含表 G.7 列出的方法。其中，a 是 vector 容器，n 是 x::size_type 型整数。

表 G.7 vector 的其他操作

操　作	描　述
a.capacity( )	返回在不要求重新分配内存的情况下，矢量能存储的元素总量
a.reserve(n)	提醒 a 对象：至少需要存储 n 个元素的内存。调用该方法后，容量至少为 n 个元素。如果 n 大于当前的容量，则需要重新分配内存。如果 n 大于 a.max_size( )，该方法将引发 length_error 异常

模板 list 还包含表 G.8 列出的方法。其中，a 和 b 是 list 容器；T 是存储在链表中的类型，如 int；t 是类型为 T 的值；i 和 j 是输入迭代器；q2 和 p 是迭代器；q 和 q1 是可解除引用的迭代器；n 是 x::size_type 型整数。该表使用了标准的 STL 表示法[i, j)，这指的是从 i 到 j（不包括 j）的区间。

表 G.8 list 的其他操作

方　法	描　述
a.splice(p, b)	将链表 b 的内容移到链表 a 中，并将它们插在 p 之前
a.splice(p, b, i)	将 i 指向的链表 b 中的元素移到链表 a 的 p 位置之前
a.splice(p, b, i, j)	将链表 b 中[i，j)区间内的元素移到链表 a 的 p 位置之前
a.remove(const T& t)	删除链表 a 中值为 t 的所有元素
a.remove_if(Predicate pred)	如果 i 是指向链表 a 中元素的迭代器，则删除 pred(*i)为 true 的所有值（Predicate 是布尔值函数或函数对象，参见第 15 章）
a.unique( )	删除连续的相同元素组中除第一个元素之外的所有元素
a.unique(BinaryPredicate bin_pred)	删除连续的 bin_pred(*i, *(i - 1))为 true 的元素组中除第一个元素之外的所有元素（BinaryPredicate 是布尔值函数或函数对象，参见第 15 章）
a.merge(b)	使用为值类型定义的<运算符，将链表 b 与链表 a 的内容合并。如果链表 a 的某个元素与链表 b 的某个元素相同，则 a 中的元素将放在前面。合并后，链表 b 为空
a.merge(b, Compare comp)	使用 comp 函数或函数对象将链表 b 与链表 a 的内容合并。如果链表 a 的某个元素与链表 b 的某个元素相同，则链表 a 中的元素将放在前面。合并后，链表 b 为空

续表

方　法	描　述
a.sort( )	使用<运算符对链表 a 进行排序
a.sort(Compare comp)	使用 comp 函数或函数对象对链表 a 进行排序
a.reverse( )	将链表 a 中的元素顺序反转

forward_list 的操作与此类似，但由于模板类 forward_list 的迭代器不能后移，有些方法必须调整。因此，用 insert_after( )、erase_after ( )和 splice_after( )替代了 insert( )、erase( )和 splice( )，这些方法都对迭代器后面而不是前面的元素进行操作。

G.4　set 和 map 的其他操作

关联容器（集合和映射是这种容器的模型）带有模板参数 Key 和 Compare，这两个参数分别表示用来对内容进行排序的键类型和用于对键值进行比较的函数对象（被称为比较对象）。对于 set 和 multiset 容器，存储的键就是存储的值，因此键类型与值类型相同。对于 map 和 multimap 容器，存储的值（模板参数 T）与键类型（模板参数 Key）相关联，值类型为 pair<const Key, T>。关联容器有其他成员来描述这些特性，如表 G.9 所示。

表 G.9　　为关联容器定义的类型

类　型	值
X::key_type	Key，键类型
X::key_compare	Compare，默认为 less<key_type>
X::value_compare	二元谓词类型，与 set 和 multiset 的 key_compare 相同，为 map 或 multimap 容器中的 pair<const Key, T>值提供了排序功能
X::mapped_type	T，关联数据类型（仅限于 map 和 multimap）

关联容器提供了表 G.10 列出的方法。通常，比较对象不要求键相同的值是相同的；等价键（equivalent key）意味着两个值（可能相同，也可能不同）的键相同。在该表中，X 为容器类，a 是类型为 X 的对象。如果 X 使用唯一键（即为 set 或 map），则 a_uniq 将是类型为 X 的对象。如果 x 使用多个键（即为 multiset 或 multimap），则 a_eq 将是类型为 X 的对象。和前面一样，i 和 j 也是指向 value_type 元素的输入迭代器，[i, j]是一个有效的区间，p 和 q2 是指向 a 的迭代器，q 和 q1 是指向 a 的可解除引用的迭代器，[q1, q2]是有效区间，t 是 X::value_type 值（可能是一对），k 是 X::key_type 值，而 il 是 initializer_list<value_type>对象。

表 G.10　　为 set、multiset、map 和 multimap 定义的操作

操　作	描　述
X(i, j, c)	创建一个空容器，插入区间[i, j]中的元素，并将 c 用作比较对象
X a(i, j, c)	创建一个名为 a 的空容器，插入区间[i, j]中的元素，并将 c 用作比较对象
X(i, j)	创建一个空容器，插入区间[i, j]中的元素，并将 Compare( )用作比较对象
X a(i, j)	创建一个名为 a 的空容器，插入区间[i, j]中的元素，并将 Compare( )用作比较对象
X(il);	等价于 X(il.begin( ), il.end( ))
a = il	将区间[il.begin( ), il.end( )]的内容赋给 a
a.key_comp( )	返回在构造 a 时使用的比较对象
a.value_comp( )	返回一个 value_compare 对象

续表

操　作	描　述
a_uniq.insert(t)	当且仅当 a 不包含具有相同键的值时，将 t 值插入到容器 a 中。该方法返回一个 pair<iterator, bool>值。如果进行了插入，则 bool 的值为 true，否则为 false。iterator 指向键与 t 相同的元素
a_eq.insert(t)	插入 t 并返回一个指向其位置的迭代器
a.insert(p, t)	将 p 作为 insert( )开始搜索的位置，将 t 插入。如果 a 是键唯一的容器，则当且仅当 a 不包含拥有相同键的元素时，才插入；否则，将进行插入。无论是否进行了插入，该方法都将返回一个迭代器，该迭代器指向拥有相同键的位置
a.insert(i, j)	将区间[i, j]中的元素插入到 a 中
a.insert(il)	将 initializer_list il 中的元素插入到 a 中
a_uniq.emplace(args)	类似于 a_uniq.insert(t)，但使用参数列表与参数包 args 的内容匹配的构造函数
a_eq.emplace(args)	类似于 a_eq.insert(t)，但使用参数列表与参数包 args 的内容匹配的构造函数
a.emplace_hint(args)	类似于 a.insert(p, t)，但使用参数列表与参数包 args 的内容匹配的构造函数
a.erase(k)	删除 a 中键与 k 相同的所有元素，并返回删除的元素数目
a.erase(q)	删除 q 指向的元素
a.erase(q1, q2)	删除区间[q1, q2)中的元素
a.clear( )	与 erase(a.begin( ), a.end( ))等效
a.find(k)	返回一个迭代器，该迭代器指向键与 k 相同的元素；如果没有找到这样的元素，则返回 a.end( )
a.count(k)	返回键与 k 相同的元素的数量
a.lower_bound(k)	返回一个迭代器，该迭代器指向第一个键不小于 k 的元素
a.upper_bound(k)	返回一个迭代器，该迭代器指向第一个键大于 k 的元素
a.equal_range(k)	返回第一个成员为 a.lower_bound(k)，第二个成员为 a.upper_bound(k)的值对
a.operator[ ](k)	返回一个引用，该引用指向与键 k 关联的值（仅限于 map 容器）

G.4　无序关联容器（C++11）

前面说过，无序关联容器（unordered_set、unordered_multiset、unordered_map 和 unordered_multimap）使用键和哈希表，以便能够快速存取数据。下面简要地介绍这些概念。哈希函数（hash function）将键转换为索引值。例如，如果键为 string 对象，哈希函数可能将其中每个字符的数字编码相加，再计算结果除以 13 的余数，从而得到一个 0～12 的索引。而无序容器将使用 13 个桶（bucket）来存储 string，所有索引为 4 的 string 都将存储在第 4 个桶中。如果您要在容器中搜索键，将对键执行哈希函数，进而只在索引对应的桶中搜索。理想情况下，应有足够多的桶，每个桶只包含为数不多的 string。

C++11 库提供了模板 hash<key>，无序关联容器默认使用该模板。为各种整型、浮点型、指针以及一些模板类（如 string）定义了该模板的具体化。

表 G.11 列出了用于这些容器的类型。

无序关联容器的接口类似于关联容器。具体地说，表 G.10 也适用于无序关联容器，但存在如下例外：不需要方法 lower_bound( )和 upper_bound( )，构造函数 X(i, j, c) 亦如此。常规关联容器是经过排序的，这让它们能够使用表示“小于”概念的比较谓词。这种比较不适用于无序关联容器，因此它们使用基于概念“等于”的比较谓词。

表 G.11　为无序关联容器定义的类型

类　型	值
X::key_type	Key，键类型
X::key_equal	Pred，一个二元谓词，检查两个类型为 Key 的参数是否相等

续表

类　　型	值
X::hasher	Hash，一个这样的二元函数对象，即如果 hf 的类型为 Hash，k 的类型为 Key，则 hf(k) 的类型为 std::size_t
X::local_iterator	一个类型与 X::iterator 相同的迭代器，但只能用于一个桶
X::const_local_iterator	一个类型与 X::const_iterator 相同的迭代器，但只能用于一个桶
X::mapped_type	T，关联数据类型（仅限于 map 和 multimap）

除表 G.10 所示的方法外，无序关联容器还包含其他一些必不可少的方法，如表 G.12 所示。在该表中，X 为无序关联容器类，a 是类型为 X 的对象，b 可能是类型为 X 的常量对象，a_uniq 是类型为 unordered_set 或 unordered_map 的对象，a_eq 是类型为 unordered_multiset 或 unordered_multimap 的对象，hf 是类型为 hasher 的值，eq 是类型为 key_equal 的值，n 是类型为 size_type 的值，z 是类型为 float 的值。与以前一样，i 和 j 也是指向 value_type 元素的输入迭代器，[i, j]是一个有效的区间，p 和 q2 是指向 a 的迭代器，q 和 q1 是指向 a 的可解除引用迭代器，[q1, q2]是有效区间，t 是 X::value_type 值（可能是一对），k 是 X::key_type 值，而 il 是 initializer_list<value_type>对象。

表 G.12　　为无序关联容器定义的操作

操　　作	描　　述
X(n, hf, eq)	创建一个至少包含 n 个桶的空容器，并将 hf 用作哈希函数，将 eq 用作键值相等谓词。如果省略了 eq，则将 key_equal( )用作键值相等谓词；如果也省略了 hf，则将 hasher( )用作哈希函数
X a(n, hf, eq)	创建一个名为 a 的空容器，它至少包含 n 个桶，并将 hf 用作哈希函数，将 eq 用作键值相等谓词。如果省略 eq，则将 key_equal( )用作键值相等谓词；如果也省略了 hf，则将 hasher( )用作哈希函数
X(i, j, n, hf, eq)	创建一个至少包含 n 个桶的空容器，将 hf 用作哈希函数，将 eq 用作键值相等谓词，并插入区间[i, j]中的元素。如果省略了 eq，将 key_equal( )用作键值相等谓词；如果省略了 hf，将 hasher( )用作哈希函数；如果省略了 n，则包含桶数不确定
X a(i, j, n, hf, eq)	创建一个名为 a 的的空容器，它至少包含 n 个桶，将 hf 用作哈希函数，将 eq 用作键值相等谓词，并插入区间[i, j]中的元素。如果省略了 eq，将 key_equal( )用作键值相等谓词；如果省略了 hf，将 hasher( )用作哈希函数；如果省略了 n，则包含桶数不确定
b.hash_function( )	返回 b 使用的哈希函数
b.key_eq( )	返回创建 b 时使用的键值相等谓词
b.bucket_count( )	返回 b 包含的桶数
b.max_bucket_count ( )	返回一个上限数，它指定了 b 最多可包含多少个桶
b.bucket(k)	返回键值为 k 的元素所属桶的索引
b.bucket_size(n)	返回索引为 n 的桶可包含的元素数
b.begin(n)	返回一个迭代器，它指向索引为 n 的桶中的第一个元素
b.end(n)	返回一个迭代器，它指向索引为 n 的桶中的最后一个元素
b.cbegin(n)	返回一个常量迭代器，它指向索引为 n 的桶中的第一个元素
b.cend(n)	返回一个常量迭代器，它指向索引为 n 的桶中的最后一个元素
b.load_factor()	返回每个桶包含的平均元素数
b.max_load_factor()	返回负载系数的最大可能取值；超过这个值后，容器将增加桶
b.max_load_factor(z)	可能修改最大负载系统，建议将它设置为 z
a.rehash(n)	将桶数调整为不小于 n，并确保 a.bucket_count( ) > a.size( ) / a.max_load_factor( )
a.reserve(n)	等价于 a.rehash(ceil(n/a.max_load_factor( ))，其中 ceil(x)返回不小于 x 的最小整数

G.5 STL 函数

STL 算法库（由头文件 algorithm 和 numeric 支持）提供了大量基于迭代器的非成员模板函数。正如第 16 章介绍的，选择的模板参数名指出了特定参数应模拟的概念。例如，ForwardIterator 用于指出，参数至少应模拟正向迭代器的要求；Predicate 用于指出，参数应是一个接受一个参数并返回 bool 值的函数对象。C++标准将算法分成 4 组：非修改式序列操作、修改式序列操作、排序和相关运算符以及数值操作（C++11 将数值操作从 STL 移到了 numeric 库中，但这不影响它们的用法）。序列操作（sequence operation）表明，函数将接受两个迭代器作为参数，它们定义了要操作的区间或序列。修改式（mutating）意味着函数可以修改容器的内容。

G.5.1 非修改式序列操作

表 G.13 对非修改式序列操作进行了总结。这里没有列出参数，而重载函数只列出了一次。表后做了更详细的说明，其中包括原型。因此，可以浏览该表，以了解函数的功能，如果对某个函数非常感兴趣，则可以了解其细节。

表 G.13 非修改式序列操作

函　数	描　述
all_of( )	如果对于所有元素的谓词测试都为 true，则返回 true。这是 C++11 新增的
any_of( )	只要对于任何一个元素的谓词测试为 true，就返回 true。这是 C++11 新增的
none_of( )	如果对于所有元素的谓词测试都为 false，则返回 true。这是 C++11 新增的
for_each( )	将一个非修改式函数对象用于区间中的每个成员
find( )	在区间中查找某个值首次出现的位置
find_if( )	在区间中查找第一个满足谓词测试条件的值
find_if_not( )	在区间中查找第一个不满足谓词测试条件的值。这是 C++11 新增的
find_end( )	在序列中查找最后一个与另一个序列匹配的值。匹配时可以使用等式或二元谓词
find_first_of( )	在第二个序列中查找第一个与第一个序列的值匹配的元素。匹配时可以使用等式或二元谓词
adjacent_find	查找第一个与其后面的元素匹配的元素。匹配时可以使用等式或二元谓词
count( )	返回特定值在区间中出现的次数
count_if( )	返回特定值与区间中的值匹配的次数，匹配时使用二元谓词
mismatch( )	查找区间中第一个与另一个区间中对应元素不匹配的元素，并返回指向这两个元素的迭代器。匹配时可以使用等式或二元谓词
equal( )	如果一个区间中的每个元素都与另一个区间中的相应元素匹配，则返回 true。匹配时可以使用等式或二元谓词
is_permutation( )	如果可通过重新排列第二个区间，使得第一个区间和第二个区间对应的元素都匹配，则返回 true，否则返回 false。匹配可以是相等，也可以使用二元谓词进行判断。这是 C++11 新增的
search( )	在序列中查找第一个与另一个序列的值匹配的值。匹配时可以使用等式或二元谓词
search_n( )	查找第一个由 n 个元素组成的序列，其中每个元素都与给定值匹配。匹配时可以使用等式或二元谓词

下面更详细地讨论这些非修改型序列操作。对于每个函数，首先列出其原型，然后做简要地描述。和前面一样，迭代器对指出了区间，而选择的模板参数名指出了迭代器的类型。通常，[first, last]区间指的是从 first 到 last（不包括 last）。有些函数接受两个区间，这两个区间的容器类型可以不同。例如，可以使用 equal( )来对链表和矢量进行比较。作为参数传递的函数是函数对象，这些函数对象可以是指针（如函数名），也可以是定义了( )操作的对象。正如第 16 章介绍的，谓词是接受一个参数的布尔函数，二元谓词是接受 2 个参数的布尔函数（函数可以不是 bool 类型，只要它对于 false 返回 0，对于 true 返回非 0 值）。

1. all_of( ) (C++11)

```
template<class InputIterator, class Predicate>
bool all_of(InputIterator first, InputIterator last,
            Predicate pred);
```

如果对于区间[first, last]中的每个迭代器，pred(*i)都为 true，或者该区间为空，则函数 all_of( )返回 true；否则返回 false。

2. any_of( ) (C++11)

```
template<class InputIterator, class Predicate>
bool any_of(InputIterator first, InputIterator last,
            Predicate pred);
```

如果对于区间[first, last]中的每个迭代器，pred(*i)都为 false，或者该区间为空，则函数 any_of( )返回 false；否则返回 true。

3. none_of( ) (C++11)

```
template<class InputIterator, class Predicate>
bool none_of(InputIterator first, InputIterator last,
             Predicate pred);
```

如果对于区间[first, last]中的每个迭代器，pred(*i)都为 false，或者该区间为空，则函数 all_of( )返回 true；否则返回 false。

4. for_each( )

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last,
                  Function f);
```

for_each( )函数将函数对象 f 用于[first, last]区间中的每个元素，它也返回 f。

5. find( )

```
template<class InputIterator, class T
InputIterator find(InputIterator first, InputIterator last,
                   Const T& value);
```

find( )函数返回一个迭代器，该迭代器指向区间[first, last]中第一个值为 value 的元素；如果没有找到这样的元素，则返回 last。

6. find_if( )

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred);
```

find_if( )函数返一个迭代器，该迭代器指向[first, last]区间中第一个对其调用函数对象 pred(*i)时结果为 true 的元素；如果没有找到这样的元素，则返回 last。

7. find_if_not( )

```
template<class InputIterator, class Predicate>
InputIterator find_if_not(InputIterator first, InputIterator last,
                          Predicate pred);
```

find_if_not( )函数返一个迭代器，该迭代器指向[first, last]区间中第一个对其调用函数对象 pred(*i)时结果为 false 的元素；如果没有找到这样的元素，则返回 last。

8. find_end( )

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(
        ForwardIterator1 first1, ForwardIterator1 last1,
```

```
        ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
    class BinaryPredicate>
ForwardIterator1 find_end(
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred);
```

find_end( )函数返回一个迭代器，该迭代器指向[first1, last1] 区间中最后一个与[first2, last2] 区间的内容匹配的序列的第一个元素。第一个版本使用值类型的= =运算符来比较元素；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。如果没有找到这样的元素，则它们都返回 last1。

9. find_first_of( )

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_first_of(
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
        class BinaryPredicate>
ForwardIterator1 find_first_of(
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred);
```

find_first_of( ) 函数返回一个迭代器，该迭代器指向区间[first1, last1]中第一个与[first2, last2]区间中的任何元素匹配的元素。第一个版本使用值类型的= =运算符对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。如果没有找到这样的元素，则它们都将返回 last1。

10. adjacent_find( )

```
template<class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first,
        ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first,
        ForwardIterator last, BinaryPredicate pred);
```

adjacent_find( )函数返回一个迭代器，该迭代器指向[first1, last1] 区间中第一个与其后面的元素匹配的元素。如果没有找到这样的元素，则返回 last。第一个版本使用值类型的= =运算符来对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。

11. count( )

```
template<class InputIterator, class T>
 typename iterator_traits<InputIterator>::difference_type
  count(InputIterator first, InputIterator last, const T& value);
```

count( )函数返回[first, last)区间中与值 value 匹配的元素数目。对值进行比较时，将使用值类型的= =运算符。返回值类型为整型，它足以存储容器所能存储的最大元素数。

12. count_if( )

```
template<class InputIterator, class Predicate>
 typename iterator traits<InputIterator>::difference type
  count_if(InputIterator first, Inputiterator last,Predicate Pred>;
```

count if( )函数返回[first, last]区间中这样的元素数目，即将其作为参数传递给函数对象 pred 时，后者的返回值为 true。

13. mismatch( )

```
template<class InputIterator1, class InputIterator2>
  pair<InputIterator1, InputIterator2>
   mismatch(InputIterator1 first1,
            InputIterator1 last1, InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
                              class BinaryPredicate>
  pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1,
             InputIterator1 last1, InputIterator2 first2,
             BinaryPredicate pred);
```

每个 mismatch( )函数都在[first1, last1)区间中查找第一个与从 first2 开始的区间中相应元素不匹配的元素，并返回两个迭代器，它们指向不匹配的两个元素。如果没有发现不匹配的情况，则返回值为 pair<last1, first2 + (last1 - first1)>。第一个版本使用= =运算符来测试匹配情况；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 false，则 it1 和 it2 指向的元素不匹配。

14. equal( )

```
template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2);
template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, BinaryPredicate pred);
```

如果[first1, last1)区间中每个元素都与以 first2 开始的序列中相应元素匹配，则 equal( )函数返回 true，否则返回 false。第一个版本使用值类型的= =运算符来比较元素；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。

15. is_permutation( )（C++11）

```
template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, BinaryPredicate pred);
```

如果通过对从 first2 开始的序列进行排列，可使其与区间[first1, last1]相应的元素匹配，则函数 is_permutation( )返回 true，否则返回 false。第一个版本使用值类型的==运算符来比较元素；第二个版本使用二元谓词函数对象 pred 来比较元素，也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。

16. search( )

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(
         ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
```

```
        class BinaryPredicate>
ForwardIterator1 search(
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred);
```

search( )函数在[first1, last1]区间中搜索第一个与[first2, last2] 区间中相应的序列匹配的序列；如果没有找到这样的序列，则返回 last1。第一个版本使用值类型的= =运算符来对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素是匹配的。

17. search_n( )

```
template<class ForwardIterator, class Size, class T>
ForwardIterator  search_n(ForwardIterator first, ForwardIterator last,
                          Size count, const T& value);

template<class ForwardIterator, class Size, class T, class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                          Size count, const T& value, BinaryPredicate pred);
```

search_n( )函数在[first1, last1)区间中查找第一个与 count 个 value 组成的序列匹配的序列；如果没有找到这样的序列，则返回 last1。第一个版本使用值类型的= =运算符来对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素是匹配的。

G.5.2 修改式序列操作

表 G.14 对修改式序列操作进行了总结。其中没有列出参数，而重载函数也只列出了一次。表后做了更详细的说明，其中包括原型。因此，可以浏览该表，以了解函数的功能，如果对某个函数非常感兴趣，则可以了解其细节。

表 G.14 修改式序列操作

函　数	描　述
copy( )	将一个区间中的元素复制到迭代器指定的位置
copy_n( )	从一个迭代器指定的地方复制 n 个元素到另一个迭代器指定的地方，这是 C++11 新增的
copy_if( )	将一个区间中满足谓词测试的元素复制到迭代器指定的地方，这是 C++11 新增的
copy_backward( )	将一个区间中的元素复制到迭代器指定的地方。复制时从区间结尾开始，由后向前进行
move( )	将一个区间中的元素移到迭代器指定的地方，这是 C++11 新增的
move_backward( )	将一个区间中的元素移到迭代器指定的地方；移动时从区间结尾开始，由后向前进行。这是 C++11 新增的
swap( )	交换引用指定的位置中存储的值
swap_ranges( )	对两个区间中对应的值进行交换
iter_swap( )	交换迭代器指定的位置中存储的值
transform( )	将函数对象用于区间中的每一个元素（或区间对中的每对元素），并将返回的值复制到另一个区间的相应位置
replace( )	用另外一个值替换区间中某个值的每个实例
replace_if( )	如果用于原始值的谓词函数对象返回 true，则使用另一个值来替换区间中某个值的所有实例
replace_copy( )	将一个区间复制到另一个区间中，使用另外一个值替换指定值的每个实例
replace_copy_if( )	将一个区间复制到另一个区间，使用指定值替换谓词函数对象为 true 的每个值
fill( )	将区间中的每一个值设置为指定的值
fill_n( )	将 n 个连续元素设置为一个值
generate( )	将区间中的每个值设置为生成器的返回值，生成器是一个不接受任何参数的函数对象
generate_n( )	将区间中的前 n 个值设置为生成器的返回值，生成器是一个不接受任何参数的函数对象
remove( )	删除区间中指定值的所有实例，并返回一个迭代器，该迭代器指向得到的区间的超尾
remove_if( )	将谓词对象返回 true 的值从区间中删除，并返回一个迭代器，该迭代器指向得到的区间的超尾

续表

函　　数	描　　述
remove_copy( )	将一个区间中的元素复制到另一个区间中，复制时忽略与指定值相同的元素
remove_copy_if( )	将一个区间中的元素复制到另一个区间中，复制时忽略谓词函数对象返回 true 的元素
unique( )	将区间内两个或多个相同元素组成的序列压缩为一个元素
unique_copy( )	将一个区间中的元素复制到另一个区间中，并将两个或多个相同元素组成的序列压缩为一个元素
reverse( )	反转区间中的元素的排列顺序
reverse_copy( )	按相反的顺序将一个区间中的元素复制到另一个区间中
rotate( )	将区间中的元素循环排列，并将元素左转
rotate_copy( )	以旋转顺序将区间中的元素复制到另一个区间中
random_shuffle( )	随机重新排列区间中的元素
shuffle( )	随机重新排列区间中的元素，使用的函数对象满足 C++11 对统一随机生成器的要求
is_partitioned( )	如果区间根据指定的谓词进行了分区，则返回 true
partition( )	将满足谓词函数对象的所有元素都放在不满足谓词函数对象的元素之前
stable_partition( )	将满足谓词函数对象的所有元素放置在不满足谓词函数对象的元素之前，每组中元素的相对顺序保持不变
partition_copy( )	将满足谓词函数对象的所有元素都复制到一个输出区间中，并将其他元素都复制到另一个输出区间中，这是 C++11 新增的
partition_point( )	对于根据指定谓词进行了分区的区间，返回一个迭代器，该迭代器指向第一个不满足该谓词的元素

下面详细地介绍这些修改型序列操作。对于每个函数，首先列出其原型，然后做简要的描述。正如前面介绍的，迭代器对指出了区间，而选择的模板参数名指出了迭代器的类型。通常，[first, last]区间指的是从 first 到 last（不包括 last）。作为参数传递的函数是函数对象，这些函数对象可以是指针，也可以是定义了( )操作的对象。正如第 16 章介绍的，谓词是接受一个参数的布尔函数，二元谓词是接受两个参数的布尔函数（函数可以不是 bool 类型，只要它对于 false 返回 0，对于 true 返回非 0 值）。另外，正如第 16 章介绍的，一元函数对象接受一个参数，而二元函数对象接受两个参数。

1. copy( )

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

copy( )函数将[first, last)区间中的元素复制到区间[result, result + (last – first))中，并返回 result + (last – first)，即指向被复制到的最后一个位置后面的迭代器。该函数要求 result 不位于[first, last)区间中，也就是说，目标不能与源重叠。

2. copy_n( )（C++11）

```
template<class InputIterator, class Size class OutputIterator>
OutputIterator copy(InputIterator first, Size n,
                    OutputIterator result);
```

函数 copy_n( )从位置 first 开始复制 n 个元素到区间[result, result + n] 中，并返回 result + n，即指向被复制到的最后一个位置后面的迭代器。该函数不要求目标和源不重叠。

3. copy_if( )（C++11）

```
template<class InputIterator, class OutputIterator,
         class Predicate>
OutputIterator copy_if(InputIterator first, InputIterator last,
        OutputIterator result, Predicate pred);
```

函数 copy_if( )将[first, last)区间中满足谓词 pred 的元素复制到区间[result, result + (last – first))中，并返回 result + (last – first)，即指向被复制到的最后一个位置后面的迭代器。该函数要求 result 不位于[first, last)区间中，也就是说，目标不能与源重叠。

4. copy_backward( )

```
template<class BidirectionalIterator1,
         class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,

BidirectionalIterator1 last, BidirectionalIterator2 result);
```

函数 copy_backward( )将[first, last)区间中的元素复制到区间[result - (last - first), result)中。复制从 last - 1 开始，该元素被复制到位置 result - 1，然后由后向前处理，直到 first。该函数返回 result - (last - first)，即指向被复制到的最后一个位置后面的迭代器。该函数要求 result 不位于[first, last)区间中。然而，由于复制是从后向前进行的，因此目标和源可能重叠。

5. move( )（C++11）

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

函数 move( )使用 std::move( )将[first, last)区间中的元素移到区间[result, result + (last – first))中，并返回 result + (last – first)，即指向被复制到的最后一个位置后面的迭代器。该函数要求 result 不位于[first, last)区间中，也就是说，目标不能与源重叠。

6. move_backward( )（C++11）

```
template<class BidirectionalIterator1,
         class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,

BidirectionalIterator1 last, BidirectionalIterator2 result);
```

函数 move_backward( ) std::move( )将[first, last)区间中的元素移到区间[result - (last - first), result)中。复制从 last - 1 开始，该元素被复制到位置 result - 1，然后由后向前处理，直到 first。该函数返回 result - (last - first)，即指向被复制到的最后一个位置后面的迭代器。该函数要求 result 不位于[first, last)区间中。然而，由于复制是从后向前进行的，因此目标和源可能重叠。

7. swap( )

```
template<class T> void swap(T& a, T& b);
```

swap( )函数对引用指定的两个位置中存储的值进行交换（C++11 将这个函数移到了头文件 utility 中）。

8. swap_ranges( )

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(
                        ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2);
```

swap_ranges( )函数将[first1, last1]区间中的值与从 first2 开始的区间中对应的值交换。这两个区间不能重叠。

9. iter_swap( )

```
template<class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

iter_swap( )函数将迭代器指定的两个位置中存储的值进行交换。

10. transform( )

```
template<class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
OutputIterator result, UnaryOperation op);
```

```
template<class InputIterator1, class InputIterator2, class OutputIterator,
         class BinaryOperation>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, OutputIterator result,
                         BinaryOperation binary_op);
```

第一个版本的 transform( )将一元函数对象 op 应用到[first, last)区间中每个元素，并将返回值赋给从 result 开始的区间中对应的元素。因此，*result 被设置为 op(*first)，依此类推。该函数返回 result + (last - first)，即目标区间的超尾值。

第二个版本的 transform( )将二元函数对象 op 应用到[first1, last1)区间和[first2, last2)区间中的每个元素，并将返回值赋给从 result 开始的区间中对应的元素。因此，*result 被设置成 op(*first1, *first2)，依此类推。该函数返回 result + (last – first)，即目标区间的超尾值。

11. replace( )

```
template<class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value);
```

replace( )函数将[first, last]中的所有 old_value 替换为 new_value。

12. replace_if( )

```
template<class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
                Predicate pred, const T& new_value);
```

replace_if( )函数使用 new_value 值替换[first, last]区间中 pred（old）为 true 的每个 old 值。

13. replace_copy( )

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
     OutputIterator result,const T& old_ value, const T& new_ value);
```

replace_copy( )函数将[first, last]区间中的元素复制到从 result 开始的区间中，但它使用 new_value 代替所有的 old_value。该函数返回 result + (last - first)，即目标区间的超尾值。

14. replace_copy_if( )

```
template<class Iterator, class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if(Iterator first, Iterator last,
    OutputIterator result, Predicate pred, const T& new_ value);
```

replace_copy_if( )函数将[first, last]区间中的元素复制到从 result 开始的区间中，但它使用 new_value 代替 pred(old)为 true 的所有 old 值。该函数返回 result + (last - first)，即目标区间的超尾值。

15. fill( )

```
template<class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);
```

fill( )函数将[first, last]区间中的每个元素都设置为 value。

16. fill_n( )

```
template<class OutputIterator, class Size, class T>
void fill_n(OutputIterator first, Size n, const T& value);
```

fill_n( )函数将从 first 位置开始的前 n 个元素都设置为 value。

17. generate( )

```
template<class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen);
```

generate( )函数将[first, last)区间中的每个元素都设置为 gen( )，其中 gen 是一个生成器函数对象，即不

接受任何参数。例如，gen 可以是一个指向 rand( )的指针。

18. generate_n( )

```
template<class OutputIterator, class Size, class Generator>
void generate_n(OutputIterator first, Size n, Generator gen);
```

generate_n( )函数将从 first 开始的区间中前 n 个元素都设置为 gen( )，其中，gen 是一个生成器函数对象，即不接受任何参数。例如，gen 可以是一个指向 rand( )的指针。

19. remove( )

```
template<class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                       const T& value);
```

remove( )函数删除[first, last)区间中所有值为 value 的元素，并返回得到的区间的超尾迭代器。该函数是稳定的，这意味着未删除的元素的顺序将保持不变。

注意：由于所有的 remove( )和 unique( )函数都不是成员函数，同时这些函数并非只能用于 STL 容器，因此它们不能重新设置容器的长度。相反，它们返回一个指示新超尾位置的迭代器。通常，被删除的元素只是被移到容器尾部。然而，对于 STL 容器，可以使用返回的迭代器和 erase( )方法来重新设置 end( )。

20. remove_if( )

```
template<class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                          Predicate pred);
```

remove_if( )函数将 pred(val)为 true 的所有 val 值从[first, last)区间删除，并返回得到的区间的超尾迭代器。该函数是稳定的，这意味着未删除的元素的顺序将保持不变。

21. remove_copy( )

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator last,
                           OutputIterator result, const T& value);
```

remove_copy( )函数将[first, last)区间中的值复制到从 result 开始的区间中，复制时将忽略 value。该函数返回得到的区间的超尾迭代器。该函数是稳定的，这意味着没有被删除的元素的顺序将保持不变。

22. remove_copy_if( )

```
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result, Predicate pred);
```

remove_copy_if( )函数将[first, last)区间中的值复制到从 result 开始的区间，但复制时忽略 pred(val)为 true 的 val。该函数返回得到的区间的超尾迭代器。该函数是稳定的，这意味着没有删除的元素的顺序将保持不变。

23. unique( )

```
template<class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                       BinaryPredicate pred);
```

unique( )函数将[first, last)区间中由两个或更多相同元素构成的序列压缩为一个元素，并返回新区间的超尾迭代器。第一个版本使用值类型的= =运算符对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素是匹配的。

24. unique_copy( )

```
template<class InputIterator, class OutputIterator>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result, BinaryPredicate pred);
```

unique_copy( )函数将[first, last)区间中的元素复制到从 result 开始的区间中，并将由两个或更多个相同元素组成的序列压缩为一个元素。该函数返回新区间的超尾迭代器。第一个版本使用值类型的= =运算符，对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素是匹配的。

25. reverse( )

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

reverse( )函数通过调用 swap(first, last - 1)等来反转[first, last]区间中的元素。

26. reverse_copy

```
template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                            BidirectionalIterator last,
                            OutputIterator result);
```

reverse_copy( )函数按相反的顺序将[first, last)区间中的元素复制到从 result 开始的区间中。这两个区间不能重叠。

27. rotate( )

```
template<class ForwardIterator>
void rotate(ForwardIterator first, ForwardIterator middle,
            ForwardIterator last);
```

rotate( )函数将[first, last)区间中的元素左旋。middle 处的元素被移到 first 处，middle + 1 处的元素被移到 first + 1 处，依此类推。middle 前的元素绕回到容器尾部，以便 first 处的元素可以紧接着 last - 1 处的元素。

28. rotate_copy( )

```
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                           ForwardIterator last, OutputIterator result);
```

rotate_copy( )函数使用为 rotate( )函数描述的旋转序列，将[first, last)区间中的元素复制到从 result 开始的区间中。

29. random_shuffle( )

```
template<class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
```

这个版本的 random_shuffle( )函数将[first, last)区间中的元素打乱。分布是一致的，即原始顺序的每种可能排列方式出现的概率相同。

30. random_shuffle( )

```
template<class RandomAccessIterator, class RandomNumberGenerator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last,
                    RandomNumberGenerator&& random);
```

这个版本的 random_shuffle( )函数将[first, last)区间中的元素打乱。函数对象 random 确定分布。假设有 n 个元素，表达式 random(n)将返回[0, n)区间中的一个值。在 C++98 中，参数 random 是一个左值引用，而在 C++11 中是一个右值引用。

31. shuffle( )

```
template<class RandomAccessIterator, class Uniform RandomNumberGenerator>
void shuffle(RandomAccessIterator first, RandomAccessIterator last,
                    UniformRandomNumberGenerator&& rgen);
```

函数 shuffle( )将[first, last)区间中的元素打乱。函数对象 rgen 确定分布，它应满足 C++11 指定的有关均匀随机数生成器的要求。假设有 n 个元素，表达式 rgen(n)将返回[0, n]区间中的一个值。

32. is_partitioned( )（C++11）

```
template<class InputIterator, class Predicate>
bool is_partitioned(InputIterator first,
                    InputIterator last, Predicate pred);
```

如果区间为空或根据 pred 进行了分区（即满足谓词 pred 的元素都在不满足该谓词的元素前面），函数 is__partitioned( )将返回 true，否则返回 false。

33. partition( )

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(BidirectionalIterator first,
                                BidirectionalIterator last,
                                Predicate pred);
```

函数 partition( )将其值 val 使得 pred(val)为 true 的元素都放在不满足该测试条件的所有元素之前。这个函数返回一个迭代器，指向最后一个使得谓词对象函数为 true 的值的后面。

34. stable_partition( )

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition(BidirectionalIterator first,
                                       BidirectionalIterator last,
                                       Predicate pred);
```

函数 stable_partition( )将其值 val 使得 pred(val)为 true 的元素都放在不满足该测试条件的所有元素之前；在这两组中，元素的相对顺序保持不变。这个函数返回一个迭代器，指向最后一个使得谓词对象函数为 true 的值的后面。

35. partition_copy( )（C++11）

```
template<class InputIterator, class OutputIterator1,
         classs OutputIterator2, class Predicate>
pair<OutputIterator1, OutputIterator2> partition_copy(
     InputIterator first, InputIterator last,
     OutputIterator1 out_true, OutputIterator2 out_false
                                Predicate pred);
```

函数 partition_copy( )将所有这样的元素都复制到从 out_true 开始的区间中，即其值 val 使得 pred(val)为 true；并将其他的元素都复制到从 out_false 开始的区间中。它返回一个 pair 对象，该对象包含两个迭代器，分别指向从 out_true 和 out_false 开始的区间的末尾。

36. partition_point( )（C++11）

```
template<class ForwardIterator, class Predicate>
ForewardIterator partition_point(ForwardIterator first,
                                 ForwardIterator last,
                                 Predicate pred);
```

函数 partition_point( )要求区间根据 pred 进行了分区。它返回一个迭代器，指向最后一个让谓词对象函数为 true 的值所在的位置。

G.5.3　排序和相关操作

表 G.15 对排序和相关操作进行了总结。其中没有列出参数，而重载函数也只列出了一次。每一个函数都有一个使用<对元素进行排序的版本和一个使用比较函数对象对元素进行排序的版本。表后做了更详细的说明，其中包括原型。因此，可以浏览该表，以了解函数的功能，如果对某个函数非常感兴趣，则可以了解其细节。

表 G.15　　排序和相关操作

函　　数	描　　述
sort( )	对区间进行排序
stable_sort( )	对区间进行排序，并保留相同元素的相对顺序
partial_sort( )	对区间进行部分排序，提供完整排序的前 n 个元素
partial_sort_copy( )	将经过部分排序的区间复制到另一个区间中
is_sorted( )	如果对区间进行了排序，则返回 true，这是 C++11 新增的
is_sorted_until( )	返回一个迭代器，指向经过排序的区间末尾，这是 C++11 新增的
nth_element( )	对于给定指向区间的迭代器，找到区间被排序时，相应位置将存储哪个元素，并将该元素放到这里
lower_bound( )	对于给定的一个值，在一个排序后的区间中找到第一个这样的位置，使得将这个值插入到这个位置前面时，不会破坏顺序
upper_bound( )	对于给定的一个值，在一个排序后的区间中找到最后一个这样的位置，使得将这个值插入到这个位置前面时，不会破坏顺序
equal_range( )	对于给定的一个值，在一个排序后的区间中找到一个最大的区间，使得将这个值插入其中的任何位置，都不会破坏顺序
binary_search( )	如果排序后的区间中包含了与给定的值相同的值，则返回 true，否则返回 false
merge( )	将两个排序后的区间合并为第三个区间
inplace_merge( )	就地合并两个相邻的、排序后的区间
includes( )	如果对于一个集合中的每个元素都可以在另外一个集合中找到，则返回 true
set_union( )	构造两个集合的并集，其中包含在任何一个集合中出现过的元素
set_intersection( )	构造两个集合的交集，其中包含在两个集合中都出现过的元素
set_difference( )	构造两个集合的差集，即包含第一个集合中且没有出现在第二个集合中的所有元素
set_symmetric_difference( )	构造由只出现在其中一个集合中的元素组成的集合
make_heap( )	将区间转换成堆
push_heap( )	将一个元素添加到堆中
pop_heap( )	删除堆中最大的元素
sort_heap( )	对堆进行排序
is_heap( )	如果区间是堆，则返回 true，这是 C++11 新增的
is_heap_until( )	返回一个迭代器，指向属于堆的区间的末尾，这是 C++11 新增的
min( )	返回两个值中较小的值，如果参数为 initializer_list，则返回最小的元素（这是 C++11 新增的）
max( )	返回两个值中较大的值，如果参数为 initializer_list，则返回最大的元素（这是 C++11 新增的）
minmax( )	返回一个 pair 对象，其中包含按递增顺序排列的两个参数；如果参数为 initializer_list，则返回 pair 对象包含最小和最大的元素。这是 C++11 新增的
min_element( )	在区间找到最小值第一次出现的位置
max_element( )	在区间找到最大值第一次出现的位置
minmax_element( )	返回一个 pair 对象，其中包含两个迭代器，它们分别指向区间中最小值第一次出现的位置和区间中最大值最后一次出现的位置。这是 C++11 新增的

续表

函　数	描　述
lexicographic_compare( )	按字母顺序比较两个序列，如果第一个序列小于第二个序列，则返回 true，否则返回 false
next-permutation( )	生成序列的下一种排列方式
previous_permutation( )	生成序列的前一种排列方式

本节中的函数使用为元素定义的<运算符或模板类型 Compare 指定的比较对象来确定两个元素的顺序。如果 comp 是一个 Compare 类型的对象，则 comp(a, b)就是 a<b 的统称，如果在排序机制中，a 在 b 之前，则返回 true。如果 a<b 返回 fasle，同时 b<a 也返回 false，则说明 a 和 b 相等。比较对象必须至少提供严格弱排序功能（strict weak ordering）。这意味着：

- 表达式 comp(a, a)一定为 false，这是“值不能比其本身小”的统称（这是严格部分）。
- 如果 comp(a, b)为 true，且 comp(b, c)也为 true，则 comp(a, c)为 true（也就是说，比较是一种可传递的关系）。
- 如果 a 与 b 等价，且 b 与 c 也等价，则 a 与 c 等价（也就是说，等价也是一种可传递的关系）。

如果想将<运算符用于整数，则等价就意味着相等，但这一结论不能推而广之。例如，可以用几个描述邮件地址的成员来定义一个结构，同时定义一个根据邮政编码对结构进行排序的 comp 对象。则邮政编码相同的地址是等价的，但它们并不相等。

下面更详细地介绍排序及相关操作。对于每个函数，首先列出其原型，然后做简要的说明。我们将这一节分成几个小节。正如前面介绍的，迭代器对指出了区间，而选择的模板参数名指出了迭代器的类型。通常，[first, last)区间指的是从 first 到 last（不包括 last）。作为参数传递的函数是函数对象，这些函数对象可以是指针，也可以是定义了( )操作的对象。正如第 16 章介绍的，谓词是接受一个参数的布尔函数，二元谓词是接受 2 个参数的布尔函数（函数可以不是 bool 类型，只要它对于 false 返回 0，对于 true 返回非 0 值）。另外，正如第 16 章介绍的，一元函数对象接受一个参数，而二元函数对象接受两个参数。

1. 排序

首先来看看排序算法。

（1）sort( )

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last,
          Compare comp);
```

sort( )函数将[first, last)区间按升序进行排序，排序时使用值类型的<运算符进行比较。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（2）stable_sort( )

```
template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

stable_sort( )函数对[first, last)区间进行排序，并保持等价元素的相对顺序不变。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（3）partial_sort( )

```
template<class RandomAccessIterator>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
```

```
                    RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
RandomAccessIterator last, Compare comp);
```

partial_sort( )函数对[first, last)区间进行部分排序。将排序后的区间的前 middle - first 个元素置于[first, middle]区间内，其余元素没有排序。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（4）partial_sort_copy( )

```
template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(InputIterator first,
                        InputIterator last,
                        RandomAccessIterator result_first,
                        RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator, class Compare>
RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
                  RandomAccessIterator result_first,
                  RandomAccessIterator result_last,
                  Compare comp);
```

partial_sort_copy( )函数将排序后的区间[first, last]中的前 n 个元素复制到区间[result_first, result_first + n]中。n 的值是 last - first 和 result_last - result_first 中较小的一个。该函数返回 result - first + n。第一个版本使用<来确定顺序，第二个版本使用比较对象 comp。

（5）is_sorted（C++11）

```
template<class ForwardIterator>
bool is_sorted(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
bool is_sorted(ForwardIterator first, ForwardIterator last
               Compare comp);
```

如果区间[first, last]是经过排序的，函数 is_sorted( )将返回 true，否则返回 false。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（6）is_sorted_until（C++11）

```
template<class ForwardIterator>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last
               Compare comp);
```

如果区间[first, last]包含的元素少于两个，函数 is_sorted_until 将返回 last；否则将返回迭代器 it，确保区间[first, it]是经过排序的。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（7）nth_element( )

```
template<class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
RandomAccessIterator last, Compare comp);
```

nth_element( )函数找到将[first, last)区间排序后的第 n 个元素，并将该元素置于第 n 个位置。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

2. 二分法搜索

二分法搜索组中的算法假设区间是经过排序的。这些算法只要求正向迭代器，但使用随机迭代器时，效率最高。

（1）lower_bound( )

```
template<class ForwardIterator, class T>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& value, Compare comp);
```

lower_bound( )函数在排序后的区间[first, last)中找到第一个这样的位置，即将 value 插入到它前面时不会破坏顺序。它返回一个指向这个位置的迭代器。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（2）upper_bound( )

```
template<class ForwardIterator, class T>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                            const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);
```

upper_bound( )函数在排序后的区间[first, last)中找到最后一个这样的位置，即将 value 插入到它前面时不会破坏顺序。它返回一个指向这个位置的迭代器。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（3）equal_range( )

```
template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator> equal_range(
    ForwardIterator first, ForwardIterator last, const T& value);
template<class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator> equal_range(
    ForwardIterator first, ForwardIterator last, const T& value,
    Compare comp);
```

equal_range( )函数在排序后的区间[first, last)区间中找到这样一个最大的子区间[it1, it2)，即将 value 插入到该区间的任何位置都不会破坏顺序。该函数返回一个由 it1 和 it2 组成的 pair。第一个版本使用<来确定顺序，第二个版本使用比较对象 comp。

（4）binary_search( )

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value);

template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value, Compare comp);
```

如果在排序后的区间[first, last]中找到与 value 等价的值，则 binary_search( )函数返回 true，否则返回 false。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

注意：前面说过，使用<进行排序时，如果 a<b 和 b<a 都为 false，则 a 和 b 等价。对于常规数字来说，等价意味着相等；但对于只根据一个成员进行排序的结构来说，情况并非如此。因此，在确保顺序不被破

坏的情况下，可插入新值的位置可能不止一个。同样，如果使用比较对象 comp 进行排序，等价意味着 comp(a, b)和 comp（b，a）都为 false（这是“如果 a 不小于 b，b 也不小于 a，则 a 与 b 等价”的统称）。

3. 合并

合并函数假设区间是经过排序的。

（1）merge( )

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator,
class Compare>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
```

merge( )函数将排序后的区间[first1, last1] 中的元素与排序后的区间[first2, last2] 中的元素进行合并，并将结果放到从 result 开始的区间中。目标区间不能与被合并的任何一个区间重叠。在两个区间中发现了等价元素时，第一个区间中的元素将位于第二个区间中的元素前面。返回值是合并的区间的超尾迭代器。第一个版本使用<来确定顺序，第二个版本使用比较对象 comp。

（2）inplace_merge( )

```
template<class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first,
     BidirectionalIterator middle, BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first,
     BidirectionalIterator middle, BidirectionalIterator last,
     Compare comp);
```

inplace_merge( )函数将两个连续的、排序后的区间——[first, middle] 和[middle, last] ——合并为一个经过排序的序列，并将其存储在[first, last] 区间中。第一个区间中的元素将位于第二个区间中的等价元素之前。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

4. 使用集合

集合操作可用于所有排序后的序列，包括集合（set）和多集合（multiset）。对于存储一个值的多个实例的容器（如 multiset）来说，定义是广义的。对于两个多集合的并集，将包含较大数目的元素实例，而交集将包含较小数目的元素实例。例如，假设多集合 A 包含了字符串“apple”7 次，多集合 B 包含该字符串 4 次。则 A 和 B 的并集将包含 7 个“apple”实例，它们的交集将包含 4 个实例。

（1）includes( )

```
template<class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2, Compare comp);
```

如果[first2, last2)区间中的每一个元素在[first1, last1)区间中都可以找到，则 includes( )函数返回 true，

否则返回 false。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（2）set_union( )

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result, Compare comp);
```

set_union( )函数构造一个由[first1, last1]区间和[first2, last2] 区间组合而成的集合，并将结果复制到 result 指定的位置。得到的区间不能与原来的任何一个区间重叠。该函数返回构造的区间的超尾迭代器。并集包含在任何一个集合（或两个集合）中出现的所有元素。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（3）set_intersection( )

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_intersection(InputIterator1 first1,
      InputIterator1 last1,InputIterator2 first2,
      InputIterator2 last2, OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_intersection(InputIterator1 first1,
      InputIterator1 last1, InputIterator2 first2,
      InputIterator2 last2, OutputIterator result,
      Compare comp);
```

set_intersection( )函数构造[first1, last1)区间和[first2, last2)区间的交集，并将结果复制到 result 指定的位置。得到的区间不能与原来的任何一个区间重叠。该函数返回构造的区间的超尾迭代器。交集包含两个集合中共有的元素。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（4）set_difference( )

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_difference(InputIterator1 first1,
      InputIterator1 last1, InputIterator2 first2,
      InputIterator2 last2, OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_difference(InputIterator1 first1,
      InputIterator1 last1, InputIterator2 first2,
      InputIterator2 last2, OutputIterator result,
      Compare comp);
```

set_difference( )函数构造[first1, last1)区间与[first2, last2)区间的差集，并将结果复制到 result 指定的位置。得到的区间不能与原来的任何一个区间重叠。该函数返回构造的区间的超尾迭代器。差集包含出现在第一个集合中，但不出现在第二个集合中的所有元素。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（5）set_symmetric_difference( )

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_symmetric_difference(
               InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, InputIterator2 last2,
               OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_symmetric_difference(
               InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, InputIterator2 last2,
                 OutputIterator result, Compare comp);
```

set_symmetric_difference( )函数构造[first1, last1)区间和[first2, last2)区间的对称（symmetric）差集，并将结果复制到 result 指定的位置。得到的区间不能与原来的任何一个区间重叠。该函数返回构造的区间的超尾迭代器。对称差集包含出现在第一个集合中，但不出现在第二个集合中，或者出现在第二个集合中，但不出现在第一个集合中的所有元素。第一个版本使用<来确定顺序，第二个版本使用比较对象 comp。

5. 使用堆

堆（heap）是一种常用的数据形式，具有这样特征：第一个元素是最大的。每当第一个元素被删除或添加了新元素时，堆都可能需要重新排列，以确保这一特征。设计堆是为了提高这两种操作的效率。

（1）make_heap( )

```
template<class RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

make_heap( )函数将[first, last)区间构造成一个堆。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（2）push_heap( )

```
template<class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

push_heap( )函数假设[first, last – 1)区间是一个有效的堆，并将 last - 1 位置（即被假设为有效堆的区间后面的一个位置）上的值添加到堆中，使[first, last)区间成为一个有效堆。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（3）pop_heap( )

```
template<class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
```

pop_heap( )函数假设[first, last)区间是一个有效堆，并将位置 last - 1 处的值与 first 处的值进行交换，使[first, last – 1]区间成为一个有效堆。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（4）sort_heap( )

```
template<class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

sort_heap( )函数假设[first, last)区间是一个有效堆，并对其进行排序。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（5）is_heap( )（C++11）

```
template<class RandomAccessIterator>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last
               Compare comp);
```

如果区间[first, last)是一个有效的堆，函数 is_heap( )将返回 true，否则返回 false。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（6）is_heap_until( )（C++11）

```
template<class RandomAccessIterator>
RandomAccessIterator is_heap_until(RandomAccessIterator first,
                                   RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
RandomAccessIterator is_heap_until(
              RandomAccessIterator first, RandomAccessIterator last
              Compare comp);
```

如果区间[first, last)包含的元素少于两个，则返回 last；否则返回迭代器 it，而区间[first, it)是一个有效的堆。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

6. 查找最小和最大值

最小函数和最大函数返回两个值或值序列中的最小值和最大值。

（1）min( )

```
template<class T> const T& min(const T& a, const T& b);

template<class T, class Compare>
const T& min(const T& a, const T& b, Compare comp);
```

这些版本的 min( )函数返回两个值中较小一个；如果这两个值相等，则返回第一个值。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

```
template<class T> T min(initializer_list<T> t);

template<class T, class Compare>
T min(initializer_list<T> t), Compare comp);
```

这些版本的 min( )函数是 C++11 新增的，它返回初始化列表 t 中最小的值。如果有多个相等的值且最小，则返回第一个。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（2）max( )

```
template<class T> const T& max(const T& a, const T& b);

template<class T, class Compare>
const T& max(const T& a, const T& b, Compare comp);
```

这些版本的 max( ) 函数返回这两个值中较大的一个；如果这两个值相等，则返回第一个值。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

```
template<class T> T max(initializer_list<T> t);

template<class T, class Compare>
T max(initializer_list<T> t), Compare comp);
```

这些版本的 max( )函数是 C++11 新增的，它返回初始化列表 t 中最大的值。如果有多个相等的值且最大，则返回第一个。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（3）minmax( )（C++11）

```
template<class T>
  pair<const T&,const T&> minmax(const T& a, const T& b);

template<class T, class Compare>
  pair<const T&,const T&> minmax(const T& a, const T& b,
                                 Compare comp);
```

如果 b 小于 a，这些版本的 minmax( )函数返回 pair(b, a)，否则返回 pair(a, b)。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

```
template<class T> pair<T,T> minmax(initializer_list<T> t);

template<class T, class Compare>
pair<T,T> minmax(initializer_list<T> t), Compare comp);
```

这些版本的 minmax( )函数返回初始化列表 t 中最小元素和最大元素的拷贝。如果有多个最小的元素，则返回其中的第一个；如果有多个最大的元素，则返回其中的最后一个。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（4）min_element( )

```
template<class ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
Compare comp);
```

min_element( )函数返回这样一个迭代器，该迭代器指向[first, last)区间中第一个最小的元素。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（5）max_element( )

```
template<class ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
Compare comp);
```

max_element( )函数返回这样一个迭代器，该迭代器指向[first, last] 区间中第一个最大的元素。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（6）minmax_element( )（C++11）

```
template<class ForwardIterator>
  pair<ForwardIterator,ForwardIterator>
     minmax_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
```

```
pair<ForwardIterator,ForwardIterator>
  minmax_element(ForwardIterator first, ForwardIterator last,
                 Compare comp);
```

函数 minmax_element( )返回一个 pair 对象，其中包含两个迭代器，分别指向区间[first, last)中最小和最大的元素。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（7）lexicographical_compare( )

```
template<class InputIterator1, class InputIterator2>
bool lexicographical_compare(
     InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(
     InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     Compare comp);
```

如果[first1, last1]区间中的元素序列按字典顺序小于[first2, last2]区间中的元素序列，则 lexicographical_compare( )函数返回 true，否则返回 false。字典比较将两个序列的第一个元素进行比较，即对*first1 和*first2 进行比较。如果*first1 小于*first2，则该函数返回 true；如果*first2 小于*first1，则返回 fasle；如果相等，则继续比较两个序列中的下一个元素。直到对应的元素不相等或到达了某个序列的结尾，比较才停止。如果在到达某个序列的结尾时，这两个序列仍然是等价的，则较短的序列较小。如果两个序列等价，且长度相等，则任何一个序列都不小于另一个序列，因此函数将返回 false。第一个版本使用<来比较元素，而第二个版本使用 comp 比较对象。字典比较是按字母顺序比较的统称。

7. 排列组合

序列的排列组合是对元素重新排序。例如，由 3 个元素组成的序列有 6 种可能的排列方式，因为对于第一个位置，有 3 种选择；给第一个位置选定元素后，第二个位置有两种选择；第三个位置有 1 种选择。例如，数字 1、2 和 3 的 6 种排列如下：

```
123 132 213 232 312 321
```

通常，由 n 个元素组成的序列有 n*(n−1)*...*1 或 n!种排列。

排列函数假设按字典顺序排列各种可能的排列组合，就像前一个例子中的 6 种排列那样。这意味着，通常，在每个排列之前和之后都有一个特定的排列。例如，213 在 231 之前，312 在 231 之后。然而，第一个排列（如示例中的 123）前面没有其他排列，而最后一个排列（321）后面没有其他排列。

（1）next_permutation( )

```
template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last, Compare comp);
```

next_permutation( )函数将[first, last]区间中的序列转换为字典顺序的下一个排列。如果下一个排列存在，则该函数返回 true；如果下一个排列不存在（即区间中包含的是字典顺序的最后一个排列），则该函数返回 false，并将区间转换为字典顺序的第一个排列。第一个版本使用<来确定顺序，而第二个版本则使用 comp 比较对象。

（2）prev_permutation( )

```
template<class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first,
```

```
                    BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first,
                    BidirectionalIterator last, Compare comp);
```

previous_permutation( )函数将[first, last] 区间中的序列转换为字典顺序的前一个序列。如果前一个排列存在，则该函数返回 true；如果前一个序列不存在（即区间包含的是字典顺序的第一个排列），则该函数返回 false，并将该区间转换为字典顺序的最后一个排列。第一个版本使用<来确定顺序，而第二个版本则使用 comp 比较对象。

G.5.4　数值运算

表 G.16 对数值运算进行了总结，这些操作是由头文件 numeric 描述的。其中没有列出参数，而重载函数也只列出了一次。每一个函数都有一个使用<对元素进行排序的版本和一个使用比较函数对象对元素进行排序的版本。表后做了更详细的说明，其中包括原型。因此，可以浏览该表，以了解函数的功能；如果对某个函数非常感兴趣，则可以了解其细节。

表 G.16　　数值运算

函　　数	描　　述
accumulate( )	计算区间中的值的总和
inner_product( )	计算 2 个区间的内部乘积
partial_sum( )	将使用一个区间计算得到的小计复制到另一个区间中
adjacent_difference( )	将使用一个区间的元素计算得到的相邻差集复制到另一个区间中
iota( )	将使用运算符++生成的一系列相邻的值赋给一个区间中的元素，这是 C++11 新增的

1. accumulate( )

```
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);

template <class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init,
             BinaryOperation binary_op);
```

accumulate( )函数将 acc 的值初始化为 init，然后按顺序对[first, last]区间中的每一个迭代器 i 执行 acc = acc + * i（第一个版本）或 acc = binary_op(acc, *i)（第二个版本）。然后返回 acc 的值。

2. inner_product( )

```
template <class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init);

template <class InputIterator1, class InputIterator2, class T,
class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init,
                BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);
```

inner_product( )函数将 acc 的值初始化为 init，然后按顺序对[first1, last1]区间中的每一个迭代器 i 和[first2, first2 + (last1−first1)]区间中对应的迭代器 j 执行 acc = * i * * j（第一个版本）或 acc = binary_op(*i, *j)（第二个版本）。也就是说，该函数对每个序列的第一个元素进行计算，得到一个值，然后对每个序列中的

第二个元素进行计算，得到另一个值，依此类推，直到到达第一个序列的结尾（因此，第二个序列至少应同第一个序列一样长）。然后，函数返回 acc 的值。

3. partial_sum( )

```
template <class InputIterator, class OutputIterator>
OutputIterator partial_sum(InputIterator first, InputIterator last,
                           OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator partial_sum(InputIterator first, InputIterator last,
                           OutputIterator result,
                           BinaryOperation binary_op);
```

partial_sum( )函数将*first 赋给*result，将*first + *(first + 1)赋给*(result + 1)（第一个版本），或者将 binary_op(*first, *(first + 1))赋给*(result + 1)（第二个版本），依此类推。也就是说，从 result 开始的序列的第 n 个元素将包含从 first 开始的序列的前 n 个元素的总和（或 binary_op 的等价物）。该函数返回结果的超尾迭代器。该算法允许 result 等于 first，也就是说，如果需要，该算法允许使用结果覆盖原来的序列。

4. adjacent_difference( )

```
template <class InputIterator, class OutputIterator>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                   OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                   OutputIterator result,
                                   BinaryOperation binary_op);
```

adjacent_difference( )函数将*first 赋给 result(*result = *first)。目标区间中随后的位置将被赋值为源区间中相邻位置的差集（或 binary_op 等价物）。也就是说，目标区间的下一个位置(result +1)将被赋值为*(first + 1) - * first（第一个版本）或 binary_op(*(first + 1), * first)（第二个版本），依此类推。该函数返回结果的超尾迭代器。该算法允许 result 等于 first，也就是说，如果需要，该算法允许结果覆盖原来的序列。

5. iota( )（C++11）

```
template <class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last,T value);
```

函数 iota( )将 value 赋给*first，再将 value 递增（就像执行运算++value），并将结果赋给下一个元素，依次类推，直到最后一个元素。

附录 H　精选读物和网上资源

有很多有关 C++和编程的优秀图书和网上资源。下述清单只是其中的一些代表作而不是全部，因此还有很多优秀的图书和网站这里没有列出；然后该清单确实具有广泛的代表性。

H.1　精选读物

- Becker, Pete. The C++ Standard Library Extensions. Upper Saddle River, NJ：Addison-Wesley，2007。

本书讨论第一个 TR1（Technical Report）库。这是一个可选的 C++98 库，但 C++11 包含其大部分元素。涉及的主题包括无序集合模板、智能指针、正则表达式库、随机数库和元组。

- Booch, Grady, Robert A. Maksimchuk, Michael W. Engel, and Bobbi J.Young. Object-Oriented Analysis and Design, Third Edition. Upper Saddle River, NJ：Addison-Wesley, 2007。

本书介绍了 OOP 概念，讨论了 OOP 方法，并提供一些示例应用程序，示例是使用 C++编写的。

- Cline, Marshall, Greg Lomow and Mike Girou. *C++FAQ，Second Edition*（C++常见问题解答，第二版）。Reading，MA：Addison-Wesley，1998。

本书解答了多个经常问到的有关 C++语言的问题。

- Josuttis, Nicolai M. The C++ Standard Library: A Tutorial and Reference（C++标准库教程和参考手册）。Reading. MA：Addison-Wesley，1999。

本书介绍了标准模板库（STL）以及其他 C++库特性，如复数支持、区域和输入/输出流。

- Karlsson, Björn. Beyond the C++ Standard Library:An Introduction to Boost. Upper Saddle River，NJ：Addison-Wesley，2006。

顾名思义，本书探讨多个 Boost 库。

- Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition*。Upper Saddle River，NJ：Addison-Wesley，2005。

本书针对的是了解 C++的程序员，提供了 55 条规定和指南。其中一些是技术性的，如解释何时应定义复制构造函数和赋值运算符；其他一些更为通用，如对 is-a 和 has-a 关系的讨论。

- Meyers, Scott. *Effctive STL: 50 Specific Ways to Improve Your Use of the Stadard Template Library*。Reading，MA：Addison-Wesley，2001。

本书提供了选择容器和算法的指南，并讨论了使用 STL 的其他方面。

- Meyers, Scott. *More Effecitve C++: 35 New Ways to Improve Your Programs and Designs*。Reading，MA：Addison-Wesley，1996。

本书秉承了《*Effecitve C++*》的传统，对语言中的一些较模糊的问题进行了解释，介绍了实现各种目标的方法，如设计智能指针，并反映了 C++程序员在过去几年中获得的其他一些经验。

- Musser, David R, Gillmer J. Derge, and Atul Saini. STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Second Edition. Reading，MA：Addison-Wesley，2001。

要介绍并演示 STL 的功能，需要一整本书，该书可满足您的需求。

- Stroustrup, Bjarne. The C++ Programming Language. Third Edition。Reading，MA：Addison-Wesley，1997。

Stroustrup 创建了 C++，因此这是一部权威作品。不过，如果对 C++有一定的了解，将可以很容易地掌

握它。它不仅介绍了语言，而且提供了多个如何使用该语言的示例，同时讨论了 OOP 方法。随着语言的发展，这本书已有多个版本，该版本增加了对标准库元素的讨论，如 STL 和字符串。

- Stroustrup, Bjarne. *The Design and Evolution of C++*。Reading，MA：Addison-Wesley，1994。

如果想了解 C++的演进过程及其为何以这种方式演进，请阅读该书。

- Vandevoorde, David and Nocoli M. Jpsittos C++Templates: The Complete Guide。Reading，MA：Addison-Wesley，2003。

有关模板的内容很多，该参考手册做了详细介绍。

H.2 网上资源

- ISO/ANSI 2011 C++标准（ISO/IEC 14882:2011）。美国国家标准化组织（ANSI）和国际标准化组织（ISO）都提供。

ANSI 提供 PDF 格式的电子版下载（售价 381 美元），这通过下述网址订购：

http://webstore.ansi.org

ISO 通过下述网址提供该文档的 PDF 文件下载和光盘，售价都是 352 瑞士法郎：

www.iso.org

价格可能有变。

- C++FAQ Lite 站点可以回答常见问题（英语、汉语、法语、俄语和葡萄牙语），它是 Cline 等编著的一本图书的删节版本。当前的网址如下：

http://www.parashift.com/C++-faq-lite

- 在下面的新闻组，可以找到有关 C++问题的比较中肯的讨论：

group:comp.lang.C++.moderated

- 使用 Google、Bing 和其他搜索引擎可找到有关特定 C++主题的信息。

附录 I　转换为 ISO 标准 C++

您可能想将一些用 C 或老式 C++版本开发的程序转换为标准 C++，本附录提供了这方面的一些指南。其中的一些内容是关于从 C 转换为 C++的，另一些是关于从老式 C++转换为标准 C++的。

I.1　使用一些预处理器编译指令的替代品

C/C++预处理器提供了一系列的编译指令。通常，C++惯例是使用这些编译指令来管理编译过程，而避免用编译指令替换代码。例如，#include 编译指令是管理程序文件的重要组件。其他编译指令（如# ifndef 和 # endif）使得能够控制是否对特定的代码块进行编译。# pragma 编译指令使得能够控制编译器特定的编译选项。这些都是非常有帮助（有时是必不可少）的工具。但使用# define 编译指令时应谨慎。

I.1.1　使用 const 而不是#define 来定义常量

符号常量可提高代码的可读性和可维护性。常量名指出了其含义，如果要修改它的值，只需定义修改一次，然后重新编译即可。C 使用预处理器来创建常量的符号名称。

```
#define MAX_LENGTH 100
```

这样，预处理器将在编译之前对源代码执行文本置换，即用 100 替换所有的 MAX_LENGTH。

而 C++则在变量声明使用限定符 const：

```
const int MAX_LENGTH = 100;
```

这样 MAX_LENGTH 将被视为一个只读的 int 变量。

使用 const 的方法有很多优越性。首先，声明显式指明了类型。使用# define 时，必须在数字后加上各种后缀来指出除 char、int 或 double 之外的类型。例如，使用 100L 来表明 long 类型，使用 3.14F 来表明 float 类型。更重要的是，const 方法可以很方便地用于复合类型，如下例所示：

```
const int base_vals[5] = {1000, 2000, 3500, 6000, 10000};
const string ans[3] = {"yes", "no", "maybe"};
```

最后，const 标识符遵循变量的作用域规则，因此，可以创建作用域为全局、名称空间或数据块的常量。在特定函数中定义常量时，不必担心其定义会与程序的其他地方使用的全局常量冲突。例如，对于下面的代码：

```
#define n 5
const int dz = 12;
...
void fizzle()
{
    int n;
    int dz;
    ...
}
```

预处理器将把：

```
int n;
```

替换为：

```
int 5;
```

从而导致编译错误。而 fizzle( )中定义的 dz 是本地变量。另外，必要时，fizzle( )可以使用作用域解析运算符（::），以::dz 的方式访问该常量。

虽然 C++借鉴了 C 语言中的关键字 const，但 C++版本更有用。例如，对于外部 const 值，C++版本有内部链接，而不是变量和 C 中 const 所使用的默认外部链接。这意味着使用 const 的程序中的每个文件都必须定义该 const。这好像增加了工作量，但实际上，它使工作更简单。使用内部链接时，可以将 const 定义放在工程中的各种文件使用的头文件中。对于外部链接，这将导致编译错误，但对于内部链接，情况并非如此。另外，由于 const 必须在使用它的文件中定义（在该文件使用的头文件中定义也满足这样的要求），因此可以将 const 值用作数组长度参数：

```
const int MAX_LENGTH = 100;
...
double loads[MAX_LENGTH];
for (int i = 0; i < MAX_LENGTH; i++)
    loads[i] = 50;
```

这在 C 语言中是行不通的，因为定义 MAX_LENGTH 的声明可能位于一个独立的文件中，在编译时，该文件可能不可用。坦白地说，在 C 语言中，可以使用 static 限定符来创建内部链接常量。也就是说，C++通过默认使用 static，让您可以少记住一件事。

顺便说一句，修订后的 C 标准（C99）允许将 const 用作数组长度，但必须将数组作为一种新式数组——变量数组，而这不是 C++标准的一部分。

在控制何时编译头文件方面，# define 编译指令仍然很有帮助：

```
// blooper.h
#ifndef _BLOOPER_H_
#define _BLOOPER_H_
// code goes here
#endif
```

但对于符号常量，习惯上还是使用 const，而不是#define。另一个好方法——尤其是在有一组相关的整型常量时——是使用 enum：

```
enum {LEVEL1 = 1, LEVEL2 = 2, LEVEL3 = 4, LEVEL4 = 8};
```

I.1.2 使用 inline 而不是# define 来定义小型函数

在创建类似于内联函数的东西时，传统的 C 语言方式是使用一个#define 宏定义：

```
#define Cube(X) X*X*X
```

这将导致预处理器进行文本置换，将 X 替换为 Cube( )的参数：

```
y = Cube(x);       // replaced with y = x*x*x;
y = Cube(x + z++);  // replaced with x + z++*x + z++*x + z++;
```

由于预处理器使用文本置换，而不是真正地传递参数，因此使用这种宏可能导致意外的、错误的结果。要避免这种错误，可以在宏中使用大量的圆括号来确保正确的运算顺序：

```
#define Cube(X) ((X)*(X)*(X))
```

但即使这样做，也无法处理使用诸如 Z++等值的情况。

C++方法是使用关键字 inline 来标识内联函数，这种方法更可靠，因为它采用的是真正的参数传递。另外，C++内联函数可以是常规函数，也可以是类方法：

```
class dormant
{
private:
    int period;
```

```
    ...
public:
    int Period() const { return period; } // automatically inline
    ...
};
```

#define 宏的一个优点是，它是无类型的，因此将其用于任何类型，运算都是有意义的。在 C++中，可以创建内联模板来使函数独立于类型，同时传递参数。

总之，请使用 C++内联技术，而不是 C 语言中的#define 宏。

I.2　使用函数原型

实际上，您没有选择的余地。虽然在 C 语言中，原型是可选的，但在 C++中，它确实是必不可少的。请注意，在使用之前定义的函数（如内联函数）是其原型。

应尽可能在函数原型和函数头中使用 const。具体地说，对于表示不可修改的数据的指针参数和引用参数，应使用 const。这不仅使编译器能够捕获修改数据的错误，也使函数更为通用。也就是说，接受 const 指针或引用的函数能够同时处理 const 数据和非 const 数据，而不使用 const 指针或引用的函数只能处理非 const 数据。

I.3　使用类型转换

Stroustrup 对 C 语言的抱怨之一是其无规律可循的类型转换运算符。确实，类型转换通常是必需的，但标准类型转换太不严格。例如，对于下面的代码：

```
struct Doof
{
    double feeb;
    double steeb;
    char sgif[10];
};
Doof leam;
short * ps = (short *) & leam;  // old syntax
int * pi = int * (&leam);       // new syntax
```

C 语言不能防止将一种类型的指针转换为另一种完全不相关的类型的指针。

从某种意义上看，这种情况与 goto 语句相似。goto 语句的问题太灵活了，导致代码混乱。解决方法是提供更严格的、结构化程度更高的 goto 版本，来处理需要使用 goto 语句的常见任务，诸如 for 循环、while 循环和 if else 语句等语言元素应运而生。对于类型转换不严格的问题，标准 C++提供了类似的解决方案，即用严格的类型转换来处理最常见的、需要进行类型转换的情况。下面是第 15 章介绍的类型转换运算符：

- dynamic_cast;
- static_cast;
- const_cast;
- reinterpret_cast。

因此，在执行涉及指针的类型转换时，应尽可能使用上述运算符之一。这样做不但可以指出类型转换的目的，并可以检查类型转换是否是按预期那样使用的。

I.4　熟悉 C++特性

如果使用的是 malloc( )和 free( )，请改用 new 和 delete；如果是使用 setjmp( )和 longjmp( )处理错误，

则请改用 try、throw 和 catch。另外，对于表示 true 和 false 的值，应将其类型声明为 bool。

1.5 使用新的头文件

C++标准指定了头文件的新名称，请参见第 2 章。如果使用的是老式头文件，则应当改用新名称。这样做不仅仅是形式上的改变，因为新版本有时新增了特性。例如，头文件 ostream 提供了对宽字符输入和输出的支持，还提供了新的控制符，如 boolalpha 和 fixed（请参见第 17 章）。对于众多格式化选项的设置来说，这些控制符提供的接口比使用 setf( )或 iomanip 函数更简单。如果确实使用的是 setf( )，则在指定常量时，请使用 ios_base 而不是 ios，即使用 ios_base::fixed 而不是 ios::fixed。另外，新的头文件包含名称空间。

1.6 使用名称空间

名称空间有助于组织程序中使用的标识符，避免名称冲突。由于标准库是使用新的头文件组织实现的，它将名称放在 std 名称空间中，因此使用这些头文件需要处理名称空间。

出于简化的目的，本书的示例通常使用编译指令 using 来使 std 名称空间中的名称可用：

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;                    // a using-directive
```

然而，不管需要与否，都导出名称空间中的所有名称，是与名称空间的初衷背道而驰的。

稍微要好些的方法是，在函数中使用 using 编译指令，这将使名称在该函数中可用。

更好也是推荐的方法是，使用 using 声明或作用域解析运算符（::），只使程序需要的名称可用。例如，下面的代码使 cin、cout 和 endl 可用于文件的剩余部分：

```
#include <iostream>
using std::cin;                         // a using-declaration
using std::cout;
using std::endl;
```

但使用作用域解析运算符只能使名称在使用该运算符的表达式中可用：

```
cout << std::fixed << x << endl;     //using the scope resolution operator
```

这样做可能很麻烦，但可以将通用的 using 声明放在一个头文件中：

```
// mynames — a header file
using std::cin;                         // a using-declaration
using std::cout;
using std::endl;
```

还可以将通用的 using 声明放在一个名称空间中：

```
// mynames — a header file
#include <iostream>

namespace io
{
    using std::cin;
    using std::cout;
    using std::endl;
}

namespace formats
{
```

```
    using std::fixed;
    using std::scientific;
    using std:boolalpha;
}
```

这样，程序可以包含该文件，并使用所需的名称空间：

```
#include "mynames"
using namespace io;
```

I.7 使用智能指针

每个 new 都应与 delete 配对使用。如果使用 new 的函数由于引发异常而提前结束，将导致问题。正如第 15 章介绍的，使用 autoptr 对象跟踪 new 创建的对象将自动完成 delete 操作。C++11 新增的 unique_ptr 和 shared_ptr 提供了更佳的替代方案。

I.8 使用 string 类

传统的 C 风格字符串深受不是真正的类型之苦。可以将字符串存储在字符数组中，也可以将字符数组初始化为字符串。但不能使用赋值运算符将字符串赋给字符数组，而必须使用 strcpy( )或 strncpy( )。不能使用关系运算符来比较 C 风格字符串，而必须使用 strcmp( )（如果忘记了这一点，使用了>运算符，将不会出现语法错误，程序将比较字符串的地址，而不是字符串的内容）。

而 string 类（参见第 16 章和附录 F）使得能够使用对象来表示字符串，并定义了赋值运算符、关系运算符和加法运算符（用于拼接）。另外，string 类还提供了自动内存管理功能，因此通常不用担心字符串被保存前，有人可能会跨越数组边界或将字符串截短。

String 类提供了许多方便的方法。例如，可以将一个 string 对象追加到另一个对象的后面，也可以将 C 风格的字符串，甚至 char 值追加到 string 对象的后面。对于接受 C 风格字符串参数的函数，可以使用 c_str( ) 方法来返回一个适当的 char 指针。

string 类不仅提供了一组设计良好的方法来处理与字符串相关的工作（如查找子字符串），而且与 STL 兼容，因此，可以将 STL 算法用于 string 对象。

I.9 使用 STL

标准模板库（请参见第 16 章和附录 G）为许多编程需要提供了现成的解决方案，应使用它。例如，与其声明一个 double 或 string 对象数组，不如创建 vector<double>对象或 vector<string>对象。这样做的好处与使用 string 对象（而不是 C 风格字符串）相似。赋值运算符已被定义，因此可以使用赋值运算符将一个 vector 对象赋给另一个 vector 对象。可以按引用传递 vector 对象，接收这种对象的函数可以使用 size( )方法来确定 vector 对象中元素数目。内置的内存管理功能使得当使用 pushback( )方法在 vector 对象中添加元素时，其大小将自动调整。当然，还可以根据实际需要来使用其他有用的类方法和通用算法。在 C++11 中，如果长度固定的数组是更好的解决方案，可使用 array<double>或 array<string>。

如果需要链表、双端队列（或队列）、栈、常规队列、集合或映射，应使用 STL，它提供了有用的容器模板。算法库使得可以将矢量的内容轻松地复制到链表中，或将集合的内容同矢量进行比较。这种设计使得 STL 成为一个工具箱，它提供了基本部件，可以根据自己的需要进行装配。

在设计内容广泛的算法库时，效率是一个主要的设计目标，因此只需要完成少量的编程工作，便可以得到最好的结果。另外，实现算法时使用了迭代器的概念，这意味着这些算法不仅可用于 STL 容器。具体地说，它们也可用于传统数组。

附录 J　复习题答案

第 2 章复习题答案

1．它们叫作函数。

2．这将导致在最终的编译之前，使用 iostream 文件的内容替换该编译指令。

3．它使得程序可以使用 std 名称空间中的定义。

4．
```
cout << "Hello, world\n";
```
或
```
cout << "Hello, world" << endl;
```

5．int cheeses;

6．cheeses = 32;

7．cin >> cheeses;

8．cout << "We have " << cheeses << " varieties of cheese\n";

9．调用函数 froop( )时，应提供一个参数，该参数的类型为 double，而该函数将返回一个 int 值。例如，可以像下面这样使用它：
```
int gval = froop(3.14159);
```
函数 rattle( )接受一个 int 参数且没有返回值。例如，可以这样使用它：
```
rattle(37);
```
函数 prune( )不接受任何参数且返回一个 int 值。例如，可以这样使用它：
```
int residue = prune();
```

10．当函数的返回类型为 void 时，不用在函数中使用 return。然而，如果不提供返回值，则可以使用它：
```
return;
```

第 3 章复习题答案

1．有多种整型类型，可以根据特定需求选择最适合的类型。例如，可以使用 short 来存储空格，使用 long 来确保存储容量，也可以寻找可提高特定计算的速度的类型。

2．
```
short rbis = 80;            // or short int rbis = 80;
unsigned int q = 42110;     // or unsigned q = 42110;
unsigned long ants = 3000000000;
// or long long ants = 3000000000;
```

注意：不要指望 int 变量能够存储 3000000000；另外，如果系统支持通用的列表初始化，可使用它：
```
short rbis = {80};          // = is optional
unsigned int q {42110};     // could use = {42110}
long long ants {3000000000};
```

3．C++没有提供自动防止超出整型限制的功能，可以使用头文件 climits 来确定限制情况。

4．常量 33L 的类型为 long，常量 33 的类型为 int。

5．这两条语句并不真正等价，虽然对于某些系统来说，它们是等效的。最重要的是，只有在使用 ASCII

码的系统上，第一条语句才将得分设置为字母 A，而第二条语句还可用于使用其他编码的系统。其次，65 是一个 int 常量，而'A'是一个 char 常量。

6. 下面是 4 种方式：

```
char c = 88;
cout << c << endl;          // char type prints as character

cout.put(char(88));         // put() prints char as character

cout << char(88) << endl;    // new-style type cast value to char

cout << (char)88 << endl;    // old-style type cast value to char
```

7. 这个问题的答案取决于这两个类型的长度。如果 long 为 4 个字节，则没有损失。因为最大的 long 值将是 20 亿，即有 10 位数。由于 double 提供了至少 13 位有效数字，因而不需要进行任何舍入。long long 类型可提供 19 位有效数字，超过了 double 保证的 13 位有效数字。

8. a. 8 * 9 + 2 is 72 + 2 is 74
   b. 6 * 3 / 4 is 18 / 4 is 4
   c. 3 / 4 * 6 is 0 * 6 is 0
   d. 6.0 * 3 / 4 is 18.0 / 4 is 4.5
   e. 15 % 4 is 3

9. 下面的代码都可用于完成第一项任务：

```
int pos = (int) x1 + (int) x2;
int pos = int(x1) + int(x2);
```

要将它们作为 double 类型相加，再进行转换，可采取下述方式之一：

```
int pos = (int) (x1 + x2);
int pos = int(x1 + x2);
```

10. a. int
    b. float
    c. char
    d. char32_t
    e. double

第 4 章复习题答案

1. a. `char actors[30];`
   b. `short betsie[100];`
   c. `float chuck[13];`
   d. `long double dipsea[64];`

2. a. `array<char,30> actors;`
   b. `array<short, 100> betsie;`
   c. `array<float, 13> chuck;`
   d. `array<long double, 64> dipsea;`

3. int oddly[5] = {1, 3, 5, 7, 9};
4. int even = oddly[0] + oddly[4];

5. cout << ideas[1] << "\n"; // or << endl;

6. char lunch[13] = "cheeseburger";// number of characters + 1
   或者
   char lunch[] = "cheeseburger"; // let the compiler count elements

7. `string lunch = "Waldorf Salad";`
   如果没有 using 编译指令，则为：
   `std::string lunch = "Waldorf Salad";`

8.
```
struct fish {
    char kind[20];
    int weight;
    float length;
};
```

9.
```
fish petes =
{
    "trout",
    12,
    26.25
};
```

10. `enum Response {No, Yes, Maybe};`

11.
```
double * pd = &ted;
cout << *pd << "\n";
```

12.
```
float * pf = treacle;   // or = &treacle[0]
cout << pf[0] << " " << pf[9] << "\n";
                // or use *pf and *(pf + 9)
```

13. 这里假设已经包含了头文件 iostream 和 vector，并有一条 using 编译指令：

```
unsigned int size;
cout << "Enter a positive integer: ";
cin >> size;
int * dyn = new int [size];
vector<int> dv(size);
```

14. 是的，它是有效的。表达式“home of the jolly bytes”是一个字符串常量，因此，它将判定为字符串开始的地址。cout 对象将 char 地址解释为打印字符串，但类型转换(int *)将地址转换为 int 指针，然后作为地址被打印。总之，该语句打印字符串的地址，只要 int 类型足够宽，能够存储该地址。

15.
```
struct fish
{
    char kind[20];
    int weight;
    float length;
};

fish * pole = new fish;
cout << "Enter kind of fish: ";
cin >> pole->kind;
```

16. 使用 cin >>address 将使得程序跳过空白，直到找到非空白字符为止。然后它将读取字符，直到再次遇到空白为止。因此，它将跳过数字输入后的换行符，从而避免这种问题。另一方面，它只读取一个单词，而不是整行。

17.
```
#include <string>
#include <vector>
```

```
#include <array>
const int Str_num {10};  // or = 10
...
std::vector<std::string> vstr(Str_num);
std::array<std::string, Str_num> astr;
```

第 5 章复习题答案

1．输入条件循环在进入输入循环体之前将评估测试表达式。如果条件最初为 false，则循环不会执行其循环体。退出条件循环在处理循环体之后评估测试表达式。因此，即使测试表达式最初为 false，循环也将执行一次。for 和 while 循环都是输入条件循环，而 do while 循环是退出条件循环。

2．它将打印下面的内容：

```
01234
```

注意，cout<< endl;不是循环体的组成部分，因为没有大括号。

3．它将打印下面的内容：

```
0369
12
```

4．它将打印下面的内容：

```
6
8
```

5．它将打印下面的内容：

```
k = 8
```

6．使用*= 运算符最简单：

```
for (int num = 1; num <= 64; num *= 2)
    cout << num << " ";
```

7．将语句放在一对大括号中将形成一个复合语句或代码块。

8．当然，第一条语句是有效的。表达式 1，024 由两个表达式组成——1 和 024，用逗号运算符连接。值为右侧表达式的值。这是 024，八进制为 20，因此该声明将值 20 赋给 X。第二条语句也是有效的。然而，运算符优先级将导致它被判定成这样：

```
(y = 1), 024;
```

也就是说，左侧表达式将 y 设置成 1，整个表达式的值（没有使用）为 024 或 20（八进制）。

9．cin>> ch 将跳过空格、换行符和制表符，其他两种格式将读取这些字符。

第 6 章复习题答案

1．这两个版本将给出相同的答案，但 if else 版本的效率更高。例如，考虑当 ch 为空格时的情况。版本 1 对空格加 1，然后看它是否为换行符。这将浪费时间，因为程序已经知道 ch 为空格，因此它不是换行符。在这种情况下，版本 2 将不会查看字符是否为换行符。

2．++ch 和 ch + 1 得到的数值相同。但++ch 的类型为 char，将作为字符打印，而 ch + 1 是 int 类型（因为将 char 和 int 相加），将作为数字打印。

3．由于程序使用 ch = '$'，而不是 ch = ='$'，因此输入和输出将如下：

```
Hi!
H$i$!$
$Send $10 or $20 now!
S$e$n$d$ $ct1 = 9, ct2 = 9
```

在第二次打印前，每个字符都被转换为$字符。另外，表达式 ch=$的值为$字符的编码，因此它是非 0 值，因而为 true;所以每次 ct2 将被加 1。

4. a. `weight >= 115 && weight < 125`

   b. `ch == 'q' || ch == 'Q'`

   c. `x % 2 == 0 && x != 26`

   d. `x % 2 == 0 && !(x % 26 == 0)`

   e. `donation >= 1000 && donation <= 2000 || guest == 1`

   f. `(ch >= 'a' && ch <= 'z') ||(ch >= 'A' && ch <= 'Z')`

5. 不一定。例如，如果 x 为 10，则!x 为 0，!!x 为 1。然而，如果 x 为 bool 变量，则!!x 为 x。

6. `(x < 0)? -x : x`

   或

   `(x >= 0)? x : -x;`

7.
```
switch (ch)
{
    case 'A': a_grade++;
              break;
    case 'B': b_grade++;
              break;
    case 'C': c_grade++;
              break;
    case 'D': d_grade++;
              break;
    default:  f_grade++;
              break;
}
```

8. 如果使用整数标签，且用户输入了非整数（如 q），则程序将因为整数输入不能处理字符而挂起。但是，如果使用字符标签，而用户输入了整数（如 5），则字符输入将 5 作为字符处理。然后，switch 语句的 default 部分将提示输入另一个字符。

9. 下面是一个版本：

```
int line = 0;
char ch;
while (cin.get(ch) && ch != 'Q')
{
    if (ch == '\n')
        line++;
}
```

第 7 章复习题答案

1. 这 3 个步骤是定义函数、提供原型、调用函数。

2. a. `void igor(void); // or void igor()`

   b. `float tofu(int n); // or float tofu(int);`

   c. `double mpg(double miles, double gallons);`

   d. `long summation(long harray[], int size);`

   e. `double doctor(const char * str);`

   f. `void ofcourse(boss dude);`

   g. `char * plot(map *pmap);`

3.
```
void set_array(int arr[], int size, int value)
{
    for (int i = 0; i < size; i++)
        arr[i] = value;
}
```

4.
```
void set_array(int * begin, int * end, int value)
{
    for (int * pt = begin; pt != end; pt++)
        pt* = value;
}
```

5.
```
double biggest (const double foot[], int size)
{
    double max;
    if (size < 1)
    {
        cout << "Invalid array size of " << size << endl;
        cout << "Returning a value of 0\n";
        return 0;
    }
    else    // not necessary because return terminates program
    {
        max = foot[0];
        for (int i = 1; i < size; i++)
            if (foot[i] > max)
                max = foot[i];
        return max;
    }
}
```

6. 将 const 限定符用于指针，以防止指向的原始数据被修改。程序传递基本类型（如 int 或 double）时，它将按值传递，以便函数使用副本。这样，原始数据将得到保护。

7. 字符串可以存储在 char 数组中，可以用带双引号的字符串来表示，也可以用指向字符串第一个字符的指针来表示。

8.
```
int replace(char * str, char c1, char c2)
{
    int count = 0;
    while (*str)      // while not at end of string
    {
        if (*str == c1)
        {
            *str = c2;
            count++;
        }
        str++;        // advance to next character
    }
    return count;
}
```

9. 由于 C++将“pizza”解释为其第一个元素的地址，因此使用*运算符将得到第一个元素的值，即字符 p。由于 C++将“taco”解释为第一个元素的地址，因此它将“taco”[2]解释为第二个元素的值，即字符 c。换句话来说，字符串常量的行为与数组名相同。

10. 要按值传递它，只要传递结构名 glitz 即可。要传递它的地址，请使用地址运算符&glitz。按值传递将自动保护原始数据，但这是以时间和内存为代价的。按地址传递可节省时间和内存，但不能保护原始

数据，除非对函数参数使用了 const 限定符。另外，按值传递意味着可以使用常规的结构成员表示法，但传递指针则必须使用间接成员运算符。

11. `int judge (int (*pf)(const char *));`

12. a. 注意，如果 ap 是一个 applicant 结构，则 ap.credit_ratings 就是一个数组名，而 ap.credit_ratings[i] 是一个数组元素：

```
void display(applicant ap)
{
    cout << ap.name << endl;
    for (int i = 0; i < 3; i++)
        cout  << ap.credit_ratings[i] <<  endl;
}
```

b. 注意，如果 pa 是一个指向 applicant 结构的指针，则 pa->credit_ratings 就是一个数组名，而 pa->credit_ratings[i]是一个数组元素：

```
void show(const applicant * pa)
{
    cout << pa->name << endl;
    for (int i = 0; i < 3; i++)
        cout  << pa->credit_ratings[i] <<  endl;
}
```

13.
```
typedef void (*p_f1)(applicant *);
p_f1 p1 = f1;
typedef const char * (*p_f2)(const applicant *, const applicant *);
p_f2 p2 = f2;
p_f1 ap[5];
p_f2 (*pa)[10];
```

第 8 章复习题答案

1. 只有一行代码的小型、非递归函数适合作为内联函数。
2. a. void song(const char * name，int times = 1);
   b. 没有。只有原型包含默认值的信息。
   c. 是的，如果保留 times 的默认值：

```
void song(char * name = "O, My Papa", int times = 1);
```

3. 可以使用字符串"\""或字符'"'来打印引号，下面的函数演示了这两种方法。

```
#include <iostream.h>
void iquote(int n)
{
    cout << "\"" << n << "\"";
}

void iquote(double x)
{
    cout << '"' << x << '"';
}

void iquote(const char * str)
{
    cout << "\"" << str << "\"";
}
```

4. a. 该函数不应修改结构成员，所以使用 const 限定符。

```
void show_box(const box & container)
{
    cout << "Made by " << container. maker << endl;
    cout << "Height = " << container.height << endl;
    cout << "Width = " << container.width << endl;
    cout << "Length = " << container.length << endl;
    cout << "Volume = " << container.volume << endl;
}
```

b.
```
void set_volume(box & crate)
{
    crate.volume = crate.height * crate.width * crate.length;
}
```

5. 首先，将原型修改成下面这样：

```
// function to modify array object
void fill(std::array<double, Seasons> & pa);
// function that uses array object without modifying it
void show(const std::array<double, Seasons> & da);
```

注意，show( )应使用 const，以禁止修改对象。
接下来，在 main( )中，将 fill( )调用改为下面这样：

```
fill(expenses);
```

函数 show( )的调用不需要修改。
接下来，新的 fill( )应类似于下面这样：

```
void fill(std::array<double, Seasons> & pa)  // changed
{
    using namespace std;
    for (int i = 0; i < Seasons; i++)
    {
        cout << "Enter " << Snames[i] << " expenses: ";
        cin >> pa[i];     // changed
    }
}
```

注意到(*pa)[i]变成了更简单的 pa[i]。
最后，修改 show( )的函数头：

```
void show(std::array<double, Seasons> & da)
```

6. a. 通过为第二个参数提供默认值：

```
double mass(double d, double v = 1.0);
```

也可以通过函数重载：

```
double mass(double d, double v);
double mass(double d);
```

b. 不能为重复的值使用默认值，因为必须从右到左提供默认值。可以使用重载：

```
void repeat(int times, const char * str);
void repeat(const char * str);
```

c. 可以使用函数重载：

```
int average(int a, int b);
double average(double x, double y);
```

d. 不能这样做，因为两个版本的特征标将相同。

7.
```
template<class T>
T max(T t1, T t2)  // or T max(const T & t1, const T & t2)
{
    return t1 > t2? t1 : t2;
}
```

8.
```
template<> box max(box b1, box b2)
{
    return b1.volume > b2.volume? b1 : b2;
}
```

9. v1 的类型为 float，v2 的类型为 float &，v3 的类型为 float &，v4 的类型为 int，v5 的类型为 double。字面值 2.0 的类型为 double，因此表达式 2.0 * m 的类型为 double。

第 9 章复习题答案

1. a. homer 将自动成为自动变量。

b. 应该在一个文件中将 secret 定义为外部变量，并在第二个文件中使用 extern 来声明它。

c. 可以在外部定义前加上关键字 static，将 topsecret 定义为一个有内部链接的静态变量。也可在一个未命名的名称空间中进行定义。

d. 应在函数中的声明前加上关键字 static，将 beencalled 定义为一个本地静态变量。

2. using 声明使得名称空间中的单个名称可用，其作用域与 using 所在的声明区域相同。using 编译指令使名称空间中的所有名称可用。使用 using 编译指令时，就像在一个包含 using 声明和名称空间本身的最小声明区域中声明了这些名称一样。

3.
```
#include <iostream>
int main()
{
    double x;
    std::cout << "Enter value: ";
    while (! (std::cin >> x) )
    {
        std::cout << "Bad input. Please enter a number: ";
        std::cin.clear();
       while (std::cin.get() != '\n')
           continue;
    }
    std::cout << "Value = " << x << std::endl;
    return 0;
}
```

4. 下面是修改后的代码：

```
#include <iostream>
int main()
{
    using std::cin;
    using std::cout;
    using std::endl;
     double x;
     cout << "Enter value: ";
```

```
    while (! (cin >> x) )
    {
        cout << "Bad input. Please enter a number: ";
        cin.clear();
        while (cin.get() != '\n')
            continue;
    }
    cout << "Value = " << x << endl;
    return 0;
}
```

5. 可以在每个文件中包含单独的静态函数定义。或者每个文件都在未命名的名称空间中定义一个合适的 average( )函数。

6.
```
10
4
0
Other: 10, 1
another(): 10, -4
```

7.
```
1
4, 1, 2
2
2
4, 1, 2
2
```

第 10 章复习题答案

1. 类是用户定义的类型的定义。类声明指定了数据将如何存储，同时指定了用来访问和操纵这些数据的方法（类成员函数）。

2. 类表示人们可以类方法的公有接口对类对象执行的操作，这是抽象。类的数据成员可以是私有的（默认值），这意味着只能通过成员函数来访问这些数据，这是数据隐藏。实现的具体细节（如数据表示和方法的代码）都是隐藏的，这是封装。

3. 类定义了一种类型，包括如何使用它。对象是一个变量或其他数据对象（如由 new 生成的），并根据类定义被创建和使用。类和对象之间的关系同标准类型与其变量之间的关系相同。

4. 如果创建给定类的多个对象，则每个对象都有其自己的数据内存空间;但所有的对象都使用同一组成员函数（通常，方法是公有的，而数据是私有的，但这只是策略方面的问题，而不是对类的要求）。

5. 这个示例使用 char 数组来存储字符数据，但可以使用 string 类对象。

```
// #include <cstring>

// class definition
class BankAccount
{
private:
    char name[40];      // or std::string name;
    char acctnum[25];   // or std::string acctnum;
    double balance;
public:
    BankAccount(const char * client, const char * num, double bal = 0.0);
//or BankAccount(const std::string & client,
//               const std::string & num, double bal = 0.0);

    void show(void) const;
    void deposit(double cash);
```

```
    void withdraw(double cash);
};
```

6. 在创建类对象或显式调用构造函数时，类的构造函数都将被调用。当对象过期时，类的析构函数将被调用。

7. 有两种可能的解决方案（要使用 strncpy()，必须包含头文件 cstring 或 string.h;要使用 string 类，必须包含头文件 string）：

```
BankAccount::BankAccount(const char * client, const char * num, double bal)
{
    strncpy(name, client, 39);
    name[39] = '\0';
    strncpy(acctnum, num, 24);
    acctnum[24] = '\0';
    balance = bal;
}
```

或者：

```
BankAccount::BankAccount(const std::string & client,
                         const std::string & num, double bal)
{
    name = client;
    acctnum = num;
    balance = bal;
}
```

请记住，默认参数位于原型中，而不是函数定义中。

8. 默认构造函数是没有参数或所有参数都有默认值的构造函数。拥有默认构造函数后，可以声明对象，而不初始化它，即使已经定义了初始化构造函数。它还使得能够声明数组。

9.
```
// stock30.h
#ifndef STOCK30_H_
#define STOCK30_H_

class Stock
{
private:
    std::string company;
    long shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    Stock();              // default constructor
    Stock(const std::string & co, long n, double pr);
    ~Stock() {}           // do-nothing destructor
    void buy(long num, double price);
    void sell(long num, double price);
    void update(double price);
    void show() const;
    const Stock & topval(const Stock & s) const;
    int numshares() const { return shares; }
    double shareval() const { return share_val; }
    double totalval() const { return total_val; }
    const string & co_name() const { return company; }
};
```

10．this 指针是类方法可以使用的指针，它指向用于调用方法的对象。因此，this 是对象的地址，*this 是对象本身。

第 11 章复习题答案

1．下面是类定义文件的原型和方法文件的函数定义：

```
// prototype
Stonewt operator*(double mult);

// definition - let constructor do the work
Stonewt Stonewt::operator*(double mult)
{
    return Stonewt(mult * pounds);
}
```

2．成员函数是类定义的一部分，通过特定的对象来调用。成员函数可以隐式访问调用对象的成员，而无需使用成员运算符。友元函数不是类的组成部分，因此被称为直接函数调用。友元函数不能隐式访问类成员，而必须将成员运算符用于作为参数传递的对象。请比较复习题 1 和复习题 4 的答案。

3．要访问私有成员，它必须是友元，但要访问公有成员，可以不是友元。

4．下面是类定义文件的原型和方法文件的函数定义：

```
// prototype
friend Stonewt operator*(double mult, const Stonewt & s);

// definition - let constructor do the work
Stonewt operator*(double mult, const Stonewt & s)
{
    return Stonewt(mult * s.pounds);
}
```

5．下面的 5 个运算符不能重载：

- sizeof。
- .。
- .*。
- ::。
- ?:。

6．这些运算符必须使用成员函数来定义。

7．下面是一个可能的原型和定义：

```
// prototype and inline definition
operator double () {return mag;}
```

但请注意，使用 magval( )方法比定义该转换函数更符合逻辑。

第 12 章复习题答案

1．a．语法是正确的，但该构造函数没有将 str 指针初始化。该构造函数应将指针设置成 NULL 或使用 new [ ]来初始化它。

b．该构造函数没有创建新的字符串，而只是复制了原有字符串的地址。它应当使用 new [ ]和 strcpy( )。

c．它复制了字符串，但没有给它分配存储空间，应使用 new char[len + 1]来分配适当数量的内存。

2．首先，当这种类型的对象过期时，对象的成员指针指向的数据仍将保留在内存中，这将占用空间，同时不可访问，因为指针已经丢失。可以让类析构函数删除构造函数中 new 分配的内存，来解决这种问题。其次，析构函数释放这种内存后，如果程序将这样的对象初始化为另一个对象，则析构函数将试图释放这

些内存两次。这是因为将一个对象初始化为另一个对象的默认初始化，将复制指针值，但不复制指向的数据，这将使两个指针指向相同的数据。解决方法是，定义一个复制构造函数，使初始化复制指向的数据。第三，将一个对象赋给另一个对象也将导致两个指针指向相同的数据。解决方法是重载赋值运算符，使之复制数据，而不是指针。

3．C++自动提供下面的成员函数：

- 如果没有定义构造函数，将提供默认构造函数。
- 如果没有定义复制构造函数，将提供复制构造函数。
- 如果没有定义赋值运算符，将提供赋值运算符。
- 如果没有定义析构函数，将提供默认析构函数。
- 如果没有定义地址运算符，将提供地址运算符。

默认构造函数不完成任何工作，但使得能够声明数组和未初始化的对象。默认复制构造函数和默认赋值运算符使用成员赋值。默认析构函数也不完成任何工作。隐式地址运算符返回调用对象的地址（即 this 指针的值）。

4．应将 personality 成员声明为字符数组或 car 指针，或者将其声明为 String 对象或 string 对象。该声明没有将方法设置为公有的，因此会有几个小错误。下面是一种可能的解决方法，修改的地方以粗体显示：

```
#include <iostream>
#include <cstring>
using namespace std;
class nifty
{
private: // optional
    char personality[40];    // provide array size
    int talents;
public: // needed
// methods
    nifty();
    nifty(const char * s);
    friend ostream & operator<<(ostream & os, const nifty & n);
};    // note closing semicolon

nifty::nifty()
{
    personality[0] = '\0';
    talents = 0;
}

nifty::nifty(const char * s)
{
    strcpy(personality, s);
    talents = 0;
}

ostream & operator<<(ostream & os, const nifty & n)
{
    os << n.personality << '\n';
    os << n.talent << '\n';
    return os;
}
```

下面是另一种解决方案：

```
#include <iostream>
#include <cstring>
using namespace std;
class nifty
{
private: // optional
    char * personality;    // create a pointer
     int talents;
 public: // needed
 // methods
    nifty();
    nifty(const char * s);
    nifty(const nifty & n);
    ~nifty() { delete [] personality; }
    nifty & operator=(const nifty & n) const;
    friend ostream & operator<<(ostream & os, const nifty & n);
};    // note closing semicolon

nifty::nifty()
{
    personality = NULL;
    talents = 0;
}

nifty::nifty(const char * s)
{
    personality = new char [strlen(s) + 1];
    strcpy(personality, s);
    talents = 0;
}

ostream & operator<<(ostream & os, const nifty & n)
{
    os << n.personality << '\n';
    os << n.talent << '\n';
    return os;
}
```

5. a.
```
Golfer nancy; // default constructor
Golfer lulu("Little Lulu"); // Golfer(const char * name, int g)
Golfer roy("Roy Hobbs", 12); // Golfer(const char * name, int g)
Golfer * par = new Golfer; // default constructor
Golfer next = lulu; // Golfer(const Golfer &g)
Golfer hazard = "Weed Thwacker"; // Golfer(const char * name, int g)
*par = nancy; // default assignment operator
nancy = "Nancy Putter";// Golfer(const char * name, int g), then
                       // the default assignment operator
```

注意，对于语句 5 和 6，有些编译器还将调用默认的赋值运算符。

b. 类应定义一个复制数据（而不是地址）的赋值运算符。

第 13 章复习题答案

1. 基类的公有成员成为派生类的公有成员。基类的保护成员成为派生类的保护成员。基类的私有成员被继承，但不能直接访问。复习题 2 的答案提供了这些通用规定的特例。

2．不能继承构造函数、析构函数、赋值运算符和友元。

3．如果返回的类型为 void，仍可以使用单个赋值，但不能使用连锁赋值：

```
baseDMA magazine("Pandering to Glitz", 1);
baseDMA gift1, gift2, gift3;
gift1 = magazine;          // ok
gift 2 = gift3 = gift1;    // no longer valid
```

如果方法返回一个对象，而不是引用，则该方法的执行速度将有所减慢，这是因为返回语句需要复制对象。

4．按派生的顺序调用构造函数，最早的构造函数最先调用。调用析构函数的顺序正好相反。

5．是的，每个类都必须有自己的构造函数。如果派生类没有添加新成员，则构造函数可以为空，但必须存在。

6．只调用派生类方法。它取代基类定义。仅当派生类没有重新定义方法或使用作用域解析运算符时，才会调用基类方法。然而，应把将所有要重新定义的函数声明为虚函数。

7．如果派生类构造函数使用 new 或 new[ ]运算符来初始化类的指针成员，则应定义一个赋值运算符。更普遍地说，如果对于派生类成员来说，默认赋值不正确，则应定义赋值运算符。

8．当然，可以将派生类对象的地址赋给基类指针;但只有通过显式类型转换，才可以将基类对象的地址赋给派生类指针（向下转换），而使用这样的指针不一定安全。

9．是的，可以将派生类对象赋给基类对象。对于派生类中新增的数据成员都不会传递给基类对象。然而，程序将使用基类的赋值运算符。仅当派生类定义了转换运算符（即包含将基类引用作为唯一参数的构造函数）或使用基类为参数的赋值运算符时，相反方向的赋值才是可能的。

10．它可以这样做，因为 C++允许基类引用指向从该基类派生而来的任何类型。

11．按值传递对象将调用复制构造函数。由于形参是基类对象，因此将调用基类的复制构造函数。复制构造函数以基类引用为参数，该引用可以指向作为参数传递的派生对象。最终结果是，将生成一个新的基类对象，其成员对应于派生对象的基类部分。

12．按引用（而不是按值）传递对象，这样可以确保函数从虚函数受益。另外，按引用（而不是按值）传递对象可以节省内存和时间，尤其对于大型对象。按值传递对象的主要优点在于可以保护原始数据，但可以通过将引用作为 const 类型传递，来达到同样的目的。

13．如果 head( )是一个常规方法，则 ph->head( )将调用 Corporation::head( );如果 head( )是一个虚函数，则 ph->head( )将调用 PublicCorporation::head( )。

14．首先，这种情况不符合 is-a 模型，因此公有继承不适用。其次，House 中的 area( )定义隐藏了 area( )的 Kitchen 版本，因为这两个方法的特征标不同。

第 14 章复习题答案

1．

Class Bear	class PolarBear	公有，北极熊是一种熊
class Kitchen	class Home	私有，家里有厨房
class Person	class Programmer	公有，程序员是一种人
class Person	class HorseAndJockey	私有，马和驯马师的组合中包含一个人
class Person class Automobile	class Driver	人是公有的，因为司机是一个人；汽车是私有的，因为司机有一辆汽车

2．

```
Gloam::Gloam(int g, const char * s) : glip(g), fb(s) { }
Gloam::Gloam(int g, const Frabjous & fr) : glip(g), fb(fr) { }
// note: the above uses the default Frabjous copy constructor
void Gloam::tell()
{
    fb.tell();
    cout << glip << endl;
}
```

3.
```
Gloam::Gloam(int g, const char * s)
            : glip(g), Frabjous(s) { }
Gloam::Gloam(int g, const Frabjous & fr)
            : glip(g), Frabjous(fr) { }
// note: the above uses the default Frabjous copy constructor
void Gloam::tell()
{
    Frabjous::tell();
    cout << glip << endl;
}
```

4.
```
class Stack<Worker *>
{
private:
    enum {MAX = 10};          // constant specific to class
    Worker * items[MAX];      // holds stack items
    int top;                  // index for top stack item
public:
    Stack();
    Boolean isempty();
    Boolean isfull();
    Boolean push(const Worker * & item); // add item to stack
    Boolean pop(Worker * & item);        // pop top into item
};
```

5.
```
ArrayTP<string> sa;
StackTP< ArrayTP<double> > stck_arr_db;
ArrayTP< StackTP<Worker *> > arr_stk_wpr;
```

程序清单 14.18 生成 4 个模板：ArrayTP<int, 10>、ArrayTP<double, 10>、ArrayTP<int, 5>和 Array<ArrayTP<int, 5>, 10>。

6. 如果两条继承路线有相同的祖先，则类中将包含祖先成员的两个拷贝。将祖先类作为虚基类可以解决这种问题。

第 15 章复习题答案

1. a. 友元声明如下：

```
friend class clasp;
```

b. 这需要一个前向声明，以便编译器能够解释 void snip（muff&）：

```
    snip(muff &):
    class muff;    // forward declaration
    class cuff {
    public:
        void snip(muff &) { ... }
        ...
    };
    class muff {
        friend void cuff::snip(muff &);
        ...
};
```

c. 首先，cuff 类声明应在 muff 类之前，以便编译器可以理解 cuff::snip( )。其次，编译器需要 muff 的一个前向声明，以便可以理解 snip(muff &)。

```
class muff;    // forward declaration
class cuff {
public:
    void snip(muff &) { ... }
    ...
};
class muff {
    friend void cuff::snip(muff &);
    ...
};
```

2．不。为使类 A 拥有一个本身为类 B 的成员函数的友元，B 的声明必须位于 A 的声明之前。一个前向声明是不够的，因为这种声明可以告诉 A：B 是一个类;但它不能指出类成员的名称。同样，如果 B 拥有一个本身是 A 的成员函数的友元，则 A 的这个声明必须位于 B 的声明之前。这两个要求是互斥的。

3．访问类的唯一方法是通过其有接口，这意味着对于 Sauce 对象，只能调用构造函数来创建一个。其他成员（soy 和 sugar）在默认情况下是私有的。

4．假设函数 f1( )调用函数 f2( )。f2( )中的返回语句导致程序执行在函数 f1( )中调用函数 f2( )后面的一条语句。throw 语句导致程序沿函数调用的当前序列回溯，直到找到直接或间接包含对 f2( )的调用的 try 语句块为止。它可能在 f1( )中、调用 f1( )的函数中或其他函数中。找到这样的 try 语句块后，将执行下一个匹配的 catch 语句块，而不是函数调用后的语句。

5．应按从子孙到祖先的顺序排列 catch 语句块。

6. 对于示例#1，如果 pg 指向一个 Superb 对象或从 Superb 派生而来的任何类的对象，则 if 条件为 true。具体地说，如果 pg 指向 Magnificent 对象，则 if 条件也为 true。对于示例#2，仅当指向 Superb 对象时，if 条件才为 true，如果指向的是从 Superb 派生出来的对象，则 if 条件不为 true。

7. Dynamic_cast 运算符只允许沿类层次结构向上转换，而 static_cast 运算符允许向上转换和向下转换。static_cast 运算符还允许枚举类型和整型之间以及数值类型之间的转换。

第 16 章复习题答案

1.
```
#include <string>
using namespace std;
class RQ1
{
private:
    string st;        // a string object
public:
    RQ1() : st("") {}
    RQ1(const char * s) : st(s) {}
    ~RQ1() {};
// more stuff
};
```

不再需要显式复制构造函数、析构程序和赋值运算符，因为 string 对象提供了自己的内存管理功能。

2．可以将一个 string 对象赋给另一个。string 对象提供了自己的内存管理功能，所以一般不需要担心字符串超出存储容量。

3.
```
#include <string>
#include <cctype>
using namespace std;
void ToUpper(string & str)
```

```
{
    for (int i = 0; i < str.size(); i++)
        str[i] = toupper(str[i]);
}
```

4.
```
auto_ptr<int> pia= new int[20];  // wrong, use with new, not new[]
auto_ptr<string>(new string);    // wrong, no name for pointer
int rigue = 7;
auto_ptr<int>(&rigue);           // wrong, memory not allocated by new
auto_ptr dbl (new double);       // wrong, omits <double>
```

5．栈的 LIFO 特征意味着可能必须在到达所需要的球棍（club）之前删除很多球棍。

6．集合将只存储每个值的一个拷贝，因此，5 个 5 分将被存储为 1 个 5 分。

7．使用迭代器使得能够使用接口类似于指针的对象遍历不以数组方式组织的数据，如双向链表中的数据。

8．STL 方法使得可以将 STL 函数用于指向常规数组的常规指针以及指向 STL 容器类的迭代器，因此提高了通用性。

9．可以将一个 vector 对象赋给另一个。vector 管理自己的内存，因此可以将元素插入到矢量中，并让它自动调整长度。使用 at( )方法，可以自动检查边界。

10．这两个 vector 函数和 random_shuffle( )函数要求随机访问迭代器，而 list 对象只有双向迭代器。可以使用 list 模板类的 sort( )成员函数（参见附录 G），而不是通用函数来排序，但没有与 random_shuffle( )等效的成员函数。然而，可以将链表复制到矢量中，然后打乱矢量，并将结果重新复制到链表中。

第 17 章复习题答案

1．iostream 文件定义了用于管理输入和输出的类、常量和操纵符，这些对象管理用于处理 I/O 的流和缓冲区。该文件还创建了一些标准对象（cin、cout、cerr 和 clog 以及对应的宽字符对象），用于处理与每个程序相连的标准输入和输出流。

2．键盘输入生成一系列字符。输入 121 将生成 3 个字符，每个字符都由一个 1 字节的二进制码表示。要将这个值存储为 int 类型，则必须将这 3 个字符转换为 121 值的二进制表示。

3．在默认情况下，标准输出和标准错误都将输出发送给标准输出设备（通常为显示器）。然而，如果要求操作系统将输出重定向到文件，则标准输出将与文件（而不是显示器）相连，但标准错误仍与显示器相连。

4．ostream 类为每种 C++基本类型定义了一个 operator <<( )函数的版本。编译器将下面的表达式：

```
cout << spot
```

解释为：

```
cout.operator<<(spot)
```

这样，它便能够将该方法调用与具有相同参数类型的函数原型匹配。

5．可以将返回 ostream &类型的输出方法拼接。这样，通过一个对象调用方法时，将返回该对象。然后，返回对象将可以调用序列中的下一个方法。

6.
```
//rq17-6.cpp
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;
```

```
    cout << "Enter an integer: ";
    int n;
    cin >> n;
    cout << setw(15) << "base ten" << setw(15)
          << "base sixteen" << setw(15) << "base eight" << "\n";
    cout.setf(ios::showbase);    // or cout << showbase;
    cout << setw(15) << n << hex << setw(15) << n
          << oct << setw(15) << n << "\n";

    return 0;
}
```

7.
```
//rq17-7.cpp
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;
    char name[20];
    float hourly;
    float hours;

    cout << "Enter your name: ";
    cin.get(name, 20).get();
    cout << "Enter your hourly wages: ";
    cin >> hourly;
    cout << "Enter number of hours worked: ";
    cin >> hours;

    cout.setf(ios::showpoint);
    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::right, ios::adjustfield);
// or cout << showpoint << fixed << right;
    cout << "First format:\n";
    cout << setw(30) << name << ": $" << setprecision(2)
         << setw(10) << hourly << ":" << setprecision(1)
         << setw(5) << hours << "\n";
    cout << "Second format:\n";
    cout.setf(ios::left, ios::adjustfield);
    cout << setw(30) << name << ": $" << setprecision(2)
         << setw(10) << hourly << ":" << setprecision(1)
         << setw(5) << hours << "\n";

    return 0;
}
```

8．下面是输出：

```
ct1 = 5; ct2 = 9
```

该程序的前半部分忽略空格和换行符，而后半部分没有。注意，程序的后半部分从第一个 q 后面的换行符开始读取，将换行符计算在内。

9．如果输入行超过 80 个字符，ignore( )将不能正常工作。在这种情况下，它将跳过前 80 个字符。

第 18 章复习题答案

1.
```
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};  // or = {8.8}
std::string s {"What a bracing effect!"};
int k{99};
Z200 zip{200,'Z',0.67});
std::vector<int> ai {3, 9, 4, 7, 1};
```

2. r1(w)合法，形参 rx 指向 w。

r1(w+1)合法，形参 rx 指向一个临时变量，这个变量被初始化为 w+1。

r1(up(w))合法，形参 rx 指向一个临时变量，这个变量被初始化为 up(w)的返回值。

一般而言，将左值传递给 const 左值引用参数时，参数将被初始化为左值。将右值传递给函数时，const 左值引用参数将指向右值的临时拷贝。

r2(w)合法，形参 rx 指向 w。

r2(w+1)非法，因为 w+1 是一个右值。

r2(up(w))非法，因为 up(w)的返回值是一个右值。

一般而言，将左值传递给非 const 左值引用参数时，参数将被初始化为左值；但非 const 左值形参不能接受右值实参。

r3(w)非法，因为右值引用不能指向左值（如 w）。

r3(w+1)合法，rx 指向表达式 w+1 的临时拷贝。

r3(up(w))合法，rx 指向 up(w)的临时返回值。

3. a.
```
double & rx
const double & rx
const double & rx
```

非 const 左值引用与左值实参 w 匹配。其他两个实参为右值，const 左值引用可指向它们的拷贝。

b.
```
double & rx
double && rx
double && rx
```

左值引用与左值实参 w 匹配，而右值引用与两个右值实参匹配。

c.
```
const double & rx
double && rx
double && rx
```

const 左值引用与左值实参 w 匹配，而右值引用与两个右值实参匹配。

总之，非 const 左值形参与左值实参匹配，非 const 右值形参与右值实参匹配；const 左值形参可与左值或右值形参匹配，但编译器优先选择前两种方式（如果可供选择的话）。

4. 它们是默认构造函数、复制构造函数、移动构造函数、析构函数、复制赋值运算符和移动赋值运算

符。这些函数之所以特殊，是因为编译器将根据情况自动提供它们的默认版本。

5. 在转让数据所有权（而不是复制数据）可行时，可使用移动构造函数，但对于标准数组，没有转让其所有权的机制。如果 Fizzle 使用指针和动态内存分配，则可将数据的地址赋给新指针，以转让其所有权。

6.
```
#include <iostream>
#include <algorithm>
template<typename T>
  void show2(double x, T fp) {std::cout << x << " -> " << fp(x) << '\n';}
int main()
{
    show2(18.0, [](double x){return 1.8*x + 32;});
    return 0;
}
```

7.
```
#include <iostream>
#include <array>
#include <algorithm>
const int Size = 5;
template<typename T>
  void sum(std::array<double,Size> a, T& fp);
int main()
{
    double total = 0.0;
    std::array<double, Size> temp_c = {32.1, 34.3, 37.8, 35.2, 34.7};
    sum(temp_c, [&total](double w){total += w;});
    std::cout << "total: " << total << '\n';
    std::cin.get();
    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);
    }
}
```