标准库还定义了一组函数来查询这些标志位的状态。操作 good 在所有错误位均未置位的情况下返回 true，而 bad、fail 和 eof 则在对应错误位被置位时返回 true。此外，在 badbit 被置位时，fail 也会返回 true。这意味着，使用 good 或 fail 是确定流的总体状态的正确方法。实际上，我们将流当作条件使用的代码就等价于!fail()。而 eof 和 bad 操作只能表示特定的错误。 313

管理条件状态

流对象的 rdstate 成员返回一个 iostate 值，对应流的当前状态。setstate 操作将给定条件位置位，表示发生了对应错误。clear 成员是一个重载的成员（参见 6.4 节，第 206 页）：它有一个不接受参数的版本，而另一个版本接受一个 iostate 类型的参数。

clear 不接受参数的版本清除（复位）所有错误标志位。执行 clear() 后，调用 good 会返回 true。我们可以这样使用这些成员：

```
// 记住 cin 的当前状态
auto old_state = cin.rdstate();  // 记住 cin 的当前状态
cin.clear();                     // 使 cin 有效
process_input(cin);              // 使用 cin
cin.setstate(old_state);         // 将 cin 置为原有状态
```

带参数的 clear 版本接受一个 iostate 值，表示流的新状态。为了复位单一的条件状态位，我们首先用 rdstate 读出当前条件状态，然后用位操作将所需位复位来生成新的状态。例如，下面的代码将 failbit 和 badbit 复位，但保持 eofbit 不变： 314

```
// 复位 failbit 和 badbit，保持其他标志位不变
cin.clear(cin.rdstate() & ~cin.failbit & ~cin.badbit);
```

8.1.2 节练习

练习 8.1：编写函数，接受一个 istream&参数，返回值类型也是 istream&。此函数须从给定流中读取数据，直至遇到文件结束标识时停止。它将读取的数据打印在标准输出上。完成这些操作后，在返回流之前，对流进行复位，使其处于有效状态。

练习 8.2：测试函数，调用参数为 cin。

练习 8.3：什么情况下，下面的 while 循环会终止？

```
while (cin >> i) /* ... */
```

8.1.3 管理输出缓冲

每个输出流都管理一个缓冲区，用来保存程序读写的数据。例如，如果执行下面的代码

```
os << "please enter a value: ";
```

文本串可能立即打印出来，但也有可能被操作系统保存在缓冲区中，随后再打印。有了缓冲机制，操作系统就可以将程序的多个输出操作组合成单一的系统级写操作。由于设备的写操作可能很耗时，允许操作系统将多个输出操作组合为单一的设备写操作可以带来很大的性能提升。

导致缓冲刷新（即，数据真正写到输出设备或文件）的原因有很多：

- 程序正常结束，作为 main 函数的 return 操作的一部分，缓冲刷新被执行。

- 缓冲区满时，需要刷新缓冲，而后新的数据才能继续写入缓冲区。
- 我们可以使用操纵符如 endl（参见 1.2 节，第 6 页）来显式刷新缓冲区。
- 在每个输出操作之后，我们可以用操纵符 unitbuf 设置流的内部状态，来清空缓冲区。默认情况下，对 cerr 是设置 unitbuf 的，因此写到 cerr 的内容都是立即刷新的。
- 一个输出流可能被关联到另一个流。在这种情况下，当读写被关联的流时，关联到的流的缓冲区会被刷新。例如，默认情况下，cin 和 cerr 都关联到 cout。因此，读 cin 或写 cerr 都会导致 cout 的缓冲区被刷新。

315 ## 刷新输出缓冲区

我们已经使用过操纵符 endl，它完成换行并刷新缓冲区的工作。IO 库中还有两个类似的操纵符：flush 和 ends。flush 刷新缓冲区，但不输出任何额外的字符；ends 向缓冲区插入一个空字符，然后刷新缓冲区：

```
cout << "hi!" << endl;   // 输出 hi 和一个换行，然后刷新缓冲区
cout << "hi!" << flush;  // 输出 hi，然后刷新缓冲区，不附加任何额外字符
cout << "hi!" << ends;   // 输出 hi 和一个空字符，然后刷新缓冲区
```

unitbuf 操纵符

如果想在每次输出操作后都刷新缓冲区，我们可以使用 unitbuf 操纵符。它告诉流在接下来的每次写操作之后都进行一次 flush 操作。而 nounitbuf 操纵符则重置流，使其恢复使用正常的系统管理的缓冲区刷新机制：

```
cout << unitbuf;          // 所有输出操作后都会立即刷新缓冲区
// 任何输出都立即刷新，无缓冲
cout << nounitbuf;        // 回到正常的缓冲方式
```

警告：如果程序崩溃，输出缓冲区不会被刷新

如果程序异常终止，输出缓冲区是不会被刷新的。当一个程序崩溃后，它所输出的数据很可能停留在输出缓冲区中等待打印。

当调试一个已经崩溃的程序时，需要确认那些你认为已经输出的数据确实已经刷新了。否则，可能将大量时间浪费在追踪代码为什么没有执行上，而实际上代码已经执行了，只是程序崩溃后缓冲区没有被刷新，输出数据被挂起没有打印而已。

关联输入和输出流

当一个输入流被关联到一个输出流时，任何试图从输入流读取数据的操作都会先刷新关联的输出流。标准库将 cout 和 cin 关联在一起，因此下面语句

```
cin >> ival;
```

导致 cout 的缓冲区被刷新。

交互式系统通常应该关联输入流和输出流。这意味着所有输出，包括用户提示信息，都会在读操作之前被打印出来。

tie 有两个重载的版本（参见 6.4 节，第 206 页）：一个版本不带参数，返回指向输

出流的指针。如果本对象当前关联到一个输出流，则返回的就是指向这个流的指针，如果对象未关联到流，则返回空指针。tie 的第二个版本接受一个指向 ostream 的指针，将自己关联到此 ostream。即，x.tie(&o)将流 x 关联到输出流 o。 316

我们既可以将一个 istream 对象关联到另一个 ostream，也可以将一个 ostream 关联到另一个 ostream：

```
cin.tie(&cout);          // 仅仅是用来展示：标准库将 cin 和 cout 关联在一起
// old_tie 指向当前关联到 cin 的流（如果有的话）
ostream *old_tie = cin.tie(nullptr); // cin 不再与其他流关联
// 将 cin 与 cerr 关联；这不是一个好主意，因为 cin 应该关联到 cout
cin.tie(&cerr);          // 读取 cin 会刷新 cerr 而不是 cout
cin.tie(old_tie);        // 重建 cin 和 cout 间的正常关联
```

在这段代码中，为了将一个给定的流关联到一个新的输出流，我们将新流的指针传递给了 tie。为了彻底解开流的关联，我们传递了一个空指针。每个流同时最多关联到一个流，但多个流可以同时关联到同一个 ostream。

8.2 文件输入输出

头文件 fstream 定义了三个类型来支持文件 IO：**ifstream** 从一个给定文件读取数据，**ofstream** 向一个给定文件写入数据，以及 **fstream** 可以读写给定文件。在 17.5.3 节中（第 676 页）我们将介绍如何对同一个文件流既读又写。

这些类型提供的操作与我们之前已经使用过的对象 cin 和 cout 的操作一样。特别是，我们可以用 IO 运算符（<<和>>）来读写文件，可以用 getline（参见 3.2.2 节，第 79 页）从一个 ifstream 读取数据，包括 8.1 节中（第 278 页）介绍的内容也都适用于这些类型。

除了继承自 iostream 类型的行为之外，fstream 中定义的类型还增加了一些新的成员来管理与流关联的文件。在表 8.3 中列出了这些操作，我们可以对 fstream、ifstream 和 ofstream 对象调用这些操作，但不能对其他 IO 类型调用这些操作。

表 8.3：fstream 特有的操作

fstream fstrm;	创建一个未绑定的文件流。*fstream* 是头文件 fstream 中定义的一个类型
fstream fstrm(s);	创建一个 *fstream*，并打开名为 s 的文件。s 可以是 string 类型，或者是一个指向 C 风格字符串的指针（参见 3.5.4 节，第 109 页）。这些构造函数都是 explicit 的（参见 7.5.4 节，第 265 页）。默认的文件模式 mode 依赖于 *fstream* 的类型
fstream fstrm(s, mode);	与前一个构造函数类似，但按指定 mode 打开文件
fstrm.open(s)	打开名为 s 的文件，并将文件与 fstrm 绑定。s 可以是一个 string 或一个指向 C 风格字符串的指针。默认的文件 mode 依赖于 *fstream* 的类型。返回 void
fstrm.close()	关闭与 fstrm 绑定的文件。返回 void
fstrm.is_open()	返回一个 bool 值，指出与 fstrm 关联的文件是否成功打开且尚未关闭

317 8.2.1 使用文件流对象

当我们想要读写一个文件时，可以定义一个文件流对象，并将对象与文件关联起来。每个文件流类都定义了一个名为 open 的成员函数，它完成一些系统相关的操作，来定位给定的文件，并视情况打开为读或写模式。

创建文件流对象时，我们可以提供文件名（可选的）。如果提供了一个文件名，则 open 会自动被调用：

```
ifstream in(ifile);        // 构造一个 ifstream 并打开给定文件
ofstream out;              // 输出文件流未关联到任何文件
```

这段代码定义了一个输入流 in，它被初始化为从文件读取数据，文件名由 string 类型的参数 ifile 指定。第二条语句定义了一个输出流 out，未与任何文件关联。在新 C++ 标准中，文件名既可以是库类型 string 对象，也可以是 C 风格字符数组（参见 3.5.4 节，第 109 页）。旧版本的标准库只允许 C 风格字符数组。

C++ 11

用 fstream 代替 iostream&

我们在 8.1 节（第 279 页）已经提到过，在要求使用基类型对象的地方，我们可以用继承类型的对象来替代。这意味着，接受一个 iostream 类型引用（或指针）参数的函数，可以用一个对应的 fstream（或 sstream）类型来调用。也就是说，如果有一个函数接受一个 ostream&参数，我们在调用这个函数时，可以传递给它一个 ofstream 对象，对 istream&和 ifstream 也是类似的。

例如，我们可以用 7.1.3 节中的 read 和 print 函数来读写命名文件。在本例中，我们假定输入和输出文件的名字是通过传递给 main 函数的参数来指定的（参见 6.2.5 节，第 196 页）：

```
ifstream input(argv[1]);      // 打开销售记录文件
ofstream output(argv[2]);     // 打开输出文件
Sales_data total;             // 保存销售总额的变量
if (read(input, total)) {     // 读取第一条销售记录
    Sales_data trans;         // 保存下一条销售记录的变量
    while(read(input, trans)) {                // 读取剩余记录
        if (total.isbn() == trans.isbn())      // 检查 isbn
            total.combine(trans);              // 更新销售总额
        else {
            print(output, total) << endl;      // 打印结果
            total = trans;                     // 处理下一本书
        }
    }
    print(output, total) << endl;     // 打印最后一本书的销售额
} else                                // 文件中无输入数据
    cerr << "No data?!" << endl;
```

除了读写的是命名文件外，这段程序与 229 页的加法程序几乎是完全相同的。重要的部分是对 read 和 print 的调用。虽然两个函数定义时指定的形参分别是 istream&和 ostream&，但我们可以向它们传递 fstream 对象。

318 成员函数 open 和 close

如果我们定义了一个空文件流对象，可以随后调用 open 来将它与文件关联起来：

```
ifstream in(ifile);              // 构筑一个 ifstream 并打开给定文件
ofstream out;                    // 输出文件流未与任何文件相关联
out.open(ifile + ".copy");       // 打开指定文件
```

如果调用 open 失败，failbit 会被置位（参见 8.1.2 节，第 280 页）。因为调用 open 可能失败，进行 open 是否成功的检测通常是一个好习惯：

```
if (out)     // 检查 open 是否成功
             // open 成功，我们可以使用文件了
```

这个条件判断与我们之前将 cin 用作条件相似。如果 open 失败，条件会为假，我们就不会去使用 out 了。

一旦一个文件流已经打开，它就保持与对应文件的关联。实际上，对一个已经打开的文件流调用 open 会失败，并会导致 failbit 被置位。随后的试图使用文件流的操作都会失败。为了将文件流关联到另外一个文件，必须首先关闭已经关联的文件。一旦文件成功关闭，我们可以打开新的文件：

```
in.close();              // 关闭文件
in.open(ifile + "2");    // 打开另一个文件
```

如果 open 成功，则 open 会设置流的状态，使得 good() 为 true。

自动构造和析构

考虑这样一个程序，它的 main 函数接受一个要处理的文件列表（参见 6.2.5 节，第 196 页）。这种程序可能会有如下的循环：

```
// 对每个传递给程序的文件执行循环操作
for (auto p = argv + 1; p != argv + argc; ++p) {
    ifstream input(*p);  // 创建输出流并打开文件
    if (input) {         // 如果文件打开成功，“处理”此文件
        process(input);
    } else
        cerr << "couldn't open: " + string(*p);
} // 每个循环步 input 都会离开作用域，因此会被销毁
```

每个循环步构造一个新的名为 input 的 ifstream 对象，并打开它来读取给定的文件。像之前一样，我们检查 open 是否成功。如果成功，将文件传递给一个函数，该函数负责读取并处理输入数据。如果 open 失败，打印一条错误信息并继续处理下一个文件。

因为 input 是 while 循环的局部变量，它在每个循环步中都要创建和销毁一次（参见 5.4.1 节，第 165 页）。当一个 fstream 对象离开其作用域时，与之关联的文件会自动关闭。在下一步循环中，input 会再次被创建。

当一个 fstream 对象被销毁时，close 会自动被调用。

8.2.1 节练习

319

练习 8.4：编写函数，以读模式打开一个文件，将其内容读入到一个 string 的 vector 中，将每一行作为一个独立的元素存于 vector 中。

练习 8.5：重写上面的程序，将每个单词作为一个独立的元素进行存储。

练习 8.6：重写 7.1.1 节的书店程序（第 229 页），从一个文件中读取交易记录。将文件名作为一个参数传递给 main（参见 6.2.5 节，第 196 页）。

8.2.2 文件模式

每个流都有一个关联的**文件模式**（file mode），用来指出如何使用文件。表 8.4 列出了文件模式和它们的含义。

表 8.4：文件模式

in	以读方式打开
out	以写方式打开
app	每次写操作前均定位到文件末尾
ate	打开文件后立即定位到文件末尾
trunc	截断文件
binary	以二进制方式进行 IO

无论用哪种方式打开文件，我们都可以指定文件模式，调用 open 打开文件时可以，用一个文件名初始化流来隐式打开文件时也可以。指定文件模式有如下限制：

- 只可以对 ofstream 或 fstream 对象设定 out 模式。
- 只可以对 ifstream 或 fstream 对象设定 in 模式。
- 只有当 out 也被设定时才可设定 trunc 模式。
- 只要 trunc 没被设定，就可以设定 app 模式。在 app 模式下，即使没有显式指定 out 模式，文件也总是以输出方式被打开。
- 默认情况下，即使我们没有指定 trunc，以 out 模式打开的文件也会被截断。为了保留以 out 模式打开的文件的内容，我们必须同时指定 app 模式，这样只会将数据追加写到文件末尾；或者同时指定 in 模式，即打开文件同时进行读写操作（参见 17.5.3 节，第 676 页，将介绍对同一个文件既进行输入又进行输出的方法）。
- ate 和 binary 模式可用于任何类型的文件流对象，且可以与其他任何文件模式组合使用。

每个文件流类型都定义了一个默认的文件模式，当我们未指定文件模式时，就使用此默认模式。与 ifstream 关联的文件默认以 in 模式打开；与 ofstream 关联的文件默认以 out 模式打开；与 fstream 关联的文件默认以 in 和 out 模式打开。

320 以 out 模式打开文件会丢弃已有数据

默认情况下，当我们打开一个 ofstream 时，文件的内容会被丢弃。阻止一个 ofstream 清空给定文件内容的方法是同时指定 app 模式：

```
// 在这几条语句中，file1 都被截断
ofstream out("file1"); // 隐含以输出模式打开文件并截断文件
ofstream out2("file1", ofstream::out); // 隐含地截断文件
ofstream out3("file1", ofstream::out | ofstream::trunc);
// 为了保留文件内容，我们必须显式指定 app 模式
ofstream app("file2", ofstream::app); // 隐含为输出模式
ofstream app2("file2", ofstream::out | ofstream::app);
```

保留被 ofstream 打开的文件中已有数据的唯一方法是显式指定 app 或 in 模式。

每次调用 open 时都会确定文件模式

对于一个给定流，每当打开文件时，都可以改变其文件模式。

```
ofstream out; // 未指定文件打开模式
out.open("scratchpad"); // 模式隐含设置为输出和截断
out.close(); // 关闭 out，以便我们将其用于其他文件
out.open("precious", ofstream::app); // 模式为输出和追加
out.close();
```

第一个 open 调用未显式指定输出模式，文件隐式地以 out 模式打开。通常情况下，out 模式意味着同时使用 trunc 模式。因此，当前目录下名为 scratchpad 的文件的内容将被清空。当打开名为 precious 的文件时，我们指定了 append 模式。文件中已有的数据都得以保留，所有写操作都在文件末尾进行。

在每次打开文件时，都要设置文件模式，可能是显式地设置，也可能是隐式地设置。当程序未指定模式时，就使用默认值。

8.2.2 节练习

练习 8.7：修改上一节的书店程序，将结果保存到一个文件中。将输出文件名作为第二个参数传递给 main 函数。

练习 8.8：修改上一题的程序，将结果追加到给定的文件末尾。对同一个输出文件，运行程序至少两次，检验数据是否得以保留。

8.3 string 流

321

sstream 头文件定义了三个类型来支持内存 IO，这些类型可以向 string 写入数据，从 string 读取数据，就像 string 是一个 IO 流一样。

istringstream 从 string 读取数据，**ostringstream** 向 string 写入数据，而头文件 **stringstream** 既可从 string 读数据也可向 string 写数据。与 fstream 类型类似，头文件 sstream 中定义的类型都继承自我们已经使用过的 iostream 头文件中定义的类型。除了继承得来的操作，sstream 中定义的类型还增加了一些成员来管理与流相关联的 string。表 8.5 列出了这些操作，可以对 stringstream 对象调用这些操作，但不能对其他 IO 类型调用这些操作。

表 8.5：stringstream 特有的操作

sstream strm;	strm 是一个未绑定的 stringstream 对象。*sstream* 是头文件 sstream 中定义的一个类型
sstream strm(s);	strm 是一个 *sstream* 对象，保存 string s 的一个拷贝。此构造函数是 explicit 的（参见 7.5.4 节，第 265 页）
strm.str()	返回 strm 所保存的 string 的拷贝
strm.str(s)	将 string s 拷贝到 strm 中。返回 void

8.3.1 使用 istringstream

当我们的某些工作是对整行文本进行处理，而其他一些工作是处理行内的单个单词

时，通常可以使用 istringstream。

考虑这样一个例子，假定有一个文件，列出了一些人和他们的电话号码。某些人只有一个号码，而另一些人则有多个——家庭电话、工作电话、移动电话等。我们的输入文件看起来可能是这样的：

```
morgan 2015552368 8625550123
drew 9735550130
lee 6095550132 2015550175 8005550000
```

文件中每条记录都以一个人名开始，后面跟随一个或多个电话号码。我们首先定义一个简单的类来描述输入数据：

```
// 成员默认为公有；参见 7.2 节（第 240 页）
struct PersonInfo {
    string name;
    vector<string> phones;
};
```

类型 PersonInfo 的对象会有一个成员来表示人名，还有一个 vector 来保存此人的所有电话号码。

322 我们的程序会读取数据文件，并创建一个 PersonInfo 的 vector。vector 中每个元素对应文件中的一条记录。我们在一个循环中处理输入数据，每个循环步读取一条记录，提取出一个人名和若干电话号码：

```
string line, word;                  // 分别保存来自输入的一行和单词
vector<PersonInfo> people;          // 保存来自输入的所有记录
// 逐行从输入读取数据，直至 cin 遇到文件尾（或其他错误）
while (getline(cin, line)) {
    PersonInfo info;                // 创建一个保存此记录数据的对象
    istringstream record(line);     // 将记录绑定到刚读入的行
    record >> info.name;            // 读取名字
    while (record >> word)          // 读取电话号码
        info.phones.push_back(word); // 保持它们
    people.push_back(info);         // 将此记录追加到 people 末尾
}
```

这里我们用 getline 从标准输入读取整条记录。如果 getline 调用成功，那么 line 中将保存着从输入文件而来的一条记录。在 while 中，我们定义了一个局部 PersonInfo 对象，来保存当前记录中的数据。

接下来我们将一个 istringstream 与刚刚读取的文本行进行绑定，这样就可以在此 istringstream 上使用输入运算符来读取当前记录中的每个元素。我们首先读取人名，随后用一个 while 循环读取此人的电话号码。

当读取完 line 中所有数据后，内层 while 循环就结束了。此循环的工作方式与前面章节中读取 cin 的循环很相似，不同之处是，此循环从一个 string 而不是标准输入读取数据。当 string 中的数据全部读出后，同样会触发“文件结束”信号，在 record 上的下一个输入操作会失败。

我们将刚刚处理好的 PersonInfo 追加到 vector 中，外层 while 循环的一个循环步就随之结束了。外层 while 循环会持续执行，直至遇到 cin 的文件结束标识。

8.3.1 节练习

练习 8.9：使用你为 8.1.2 节（第 281 页）第一个练习所编写的函数打印一个 istringstream 对象的内容。

练习 8.10：编写程序，将来自一个文件中的行保存在一个 vector<string>中。然后使用一个 istringstream 从 vector 读取数据元素，每次读取一个单词。

练习 8.11：本节的程序在外层 while 循环中定义了 istringstream 对象。如果 record 对象定义在循环之外，你需要对程序进行怎样的修改？重写程序，将 record 的定义移到 while 循环之外，验证你设想的修改方法是否正确。

练习 8.12：我们为什么没有在 PersonInfo 中使用类内初始化？

8.3.2 使用 ostringstream

323

当我们逐步构造输出，希望最后一起打印时，ostringstream 是很有用的。例如，对上一节的例子，我们可能想逐个验证电话号码并改变其格式。如果所有号码都是有效的，我们希望输出一个新的文件，包含改变格式后的号码。对于那些无效的号码，我们不会将它们输出到新文件中，而是打印一条包含人名和无效号码的错误信息。

由于我们不希望输出有无效电话号码的人，因此对每个人，直到验证完所有电话号码后才可以进行输出操作。但是，我们可以先将输出内容“写入”到一个内存 ostringstream 中：

```
for (const auto &entry : people) { // 对people中每一项
    ostringstream formatted, badNums; // 每个循环步创建的对象
    for (const auto &nums : entry.phones) { // 对每个数
        if (!valid(nums)) {
            badNums << " " << nums; // 将数的字符串形式存入badNums
        } else
            // 将格式化的字符串“写入”formatted
            formatted << " " << format(nums);
    }
    if (badNums.str().empty())          // 没有错误的数
        os << entry.name << " "         // 打印名字
           << formatted.str() << endl;  // 和格式化的数
    else // 否则，打印名字和错误的数
        cerr << "input error: " << entry.name
             << " invalid number(s) " << badNums.str() << endl;
}
```

在此程序中，我们假定已有两个函数，valid 和 format，分别完成电话号码验证和改变格式的功能。程序最有趣的部分是对字符串流 formatted 和 badNums 的使用。我们使用标准的输出运算符(<<)向这些对象写入数据，但这些“写入”操作实际上转换为 string 操作，分别向 formatted 和 badNums 中的 string 对象添加字符。

8.3.2 节练习

练习 8.13：重写本节的电话号码程序，从一个命名文件而非 cin 读取数据。

练习 8.14：我们为什么将 entry 和 nums 定义为 const auto&？

324 小结

C++使用标准库类来处理面向流的输入和输出：

- iostream 处理控制台 IO
- fstream 处理命名文件 IO
- stringstream 完成内存 string 的 IO

类 fstream 和 stringstream 都是继承自类 iostream 的。输入类都继承自 istream，输出类都继承自 ostream。因此，可以在 istream 对象上执行的操作，也可在 ifstream 或 istringstream 对象上执行。继承自 ostream 的输出类也有类似情况。

每个 IO 对象都维护一组条件状态，用来指出此对象上是否可以进行 IO 操作。如果遇到了错误——例如在输入流上遇到了文件末尾，则对象的状态变为失效，所有后续输入操作都不能执行，直至错误被纠正。标准库提供了一组函数，用来设置和检测这些状态。

术语表

条件状态（condition state）可被任何流类使用的一组标志和函数，用来指出给定流是否可用。

文件模式（file mode）类 fstream 定义的一组标志，在打开文件时指定，用来控制文件如何被使用。

文件流（file stream）用来读写命名文件的流对象。除了普通的 iostream 操作，文件流还定义了 open 和 close 成员。成员函数 open 接受一个 string 或一个 C 风格字符串参数，指定要打开的文件名，它还可以接受一个可选的参数，指明文件打开模式。成员函数 close 关闭流所关联的文件，调用 close 后才可以调用 open 打开另一个文件。

fstream 用于同时读写一个相同文件的文件流。默认情况下，fstream 以 in 和 out 模式打开文件。

ifstream 用于从输入文件读取数据的文件流。默认情况下，ifstream 以 in 模式打开文件。

继承（inheritance）程序设计功能，令一个类型可以从另一个类型继承接口。类 ifstream 和 istringstream 继承自 istream，ofstream 和 ostringstream 继承自 ostream。第 15 章将介绍继承。

istringstream 用来从给定 string 读取数据的字符串流。

ofstream 用来向输出文件写入数据的文件流。默认情况下，ofstream 以 out 模式打开文件。

字符串流（string stream）用于读写 string 的流对象。除了普通的 iostream 操作外，字符串流还定义了一个名为 str 的重载成员。调用 str 的无参版本会返回字符串流关联的 string。调用时传递给它一个 string 参数，则会将字符串流与该 string 的一个拷贝相关联。

stringstream 用于读写给定 string 的字符串流。

第 9 章
顺序容器

内容

本章是第 3 章内容的扩展，完成本章的学习后，对标准库顺序容器知识的掌握就完整了。元素在顺序容器中的顺序与其加入容器时的位置相对应。标准库还定义了几种关联容器，关联容器中元素的位置由元素相关联的关键字值决定。我们将在第 11 章中介绍关联容器特有的操作。

所有容器类都共享公共的接口，不同容器按不同方式对其进行扩展。这个公共接口使容器的学习更加容易——我们基于某种容器所学习的内容也都适用于其他容器。每种容器都提供了不同的性能和功能的权衡。

326 一个容器就是一些特定类型对象的集合。**顺序容器**（sequential container）为程序员提供了控制元素存储和访问顺序的能力。这种顺序不依赖于元素的值，而是与元素加入容器时的位置相对应。与之相对的，我们将在第 11 章介绍的有序和无序关联容器，则根据关键字的值来存储元素。

标准库还提供了三种容器适配器，分别为容器操作定义了不同的接口，来与容器类型适配。我们将在本章末尾介绍适配器。

本章的内容基于 3.2 节、3.3 节和 3.4 节中已经介绍的有关容器的知识，我们假定读者已经熟悉了这几节的内容。

9.1 顺序容器概述

表 9.1 列出了标准库中的顺序容器，所有顺序容器都提供了快速顺序访问元素的能力。但是，这些容器在以下方面都有不同的性能折中：

- 向容器添加或从容器中删除元素的代价
- 非顺序访问容器中元素的代价

表 9.1：顺序容器类型

`vector`	可变大小数组。支持快速随机访问。在尾部之外的位置插入或删除元素可能很慢
`deque`	双端队列。支持快速随机访问。在头尾位置插入/删除速度很快
`list`	双向链表。只支持双向顺序访问。在 `list` 中任何位置进行插入/删除操作速度都很快
`forward_list`	单向链表。只支持单向顺序访问。在链表任何位置进行插入/删除操作速度都很快
`array`	固定大小数组。支持快速随机访问。不能添加或删除元素
`string`	与 `vector` 相似的容器，但专门用于保存字符。随机访问快。在尾部插入/删除速度快

除了固定大小的 `array` 外，其他容器都提供高效、灵活的内存管理。我们可以添加和删除元素，扩张和收缩容器的大小。容器保存元素的策略对容器操作的效率有着固有的，有时是重大的影响。在某些情况下，存储策略还会影响特定容器是否支持特定操作。

327 例如，`string` 和 `vector` 将元素保存在连续的内存空间中。由于元素是连续存储的，由元素的下标来计算其地址是非常快速的。但是，在这两种容器的中间位置添加或删除元素就会非常耗时：在一次插入或删除操作后，需要移动插入/删除位置之后的所有元素，来保持连续存储。而且，添加一个元素有时可能还需要分配额外的存储空间。在这种情况下，每个元素都必须移动到新的存储空间中。

`list` 和 `forward_list` 两个容器的设计目的是令容器任何位置的添加和删除操作都很快速。作为代价，这两个容器不支持元素的随机访问：为了访问一个元素，我们只能遍历整个容器。而且，与 `vector`、`deque` 和 `array` 相比，这两个容器的额外内存开销也很大。

`deque` 是一个更为复杂的数据结构。与 `string` 和 `vector` 类似，`deque` 支持快速

的随机访问。与 string 和 vector 一样，在 deque 的中间位置添加或删除元素的代价（可能）很高。但是，在 deque 的两端添加或删除元素都是很快的，与 list 或 forward_list 添加删除元素的速度相当。

forward_list 和 array 是新 C++标准增加的类型。与内置数组相比，array 是一种更安全、更容易使用的数组类型。与内置数组类似，array 对象的大小是固定的。因此，array 不支持添加和删除元素以及改变容器大小的操作。forward_list 的设计目标是达到与最好的手写的单向链表数据结构相当的性能。因此，forward_list 没有 size 操作，因为保存或计算其大小就会比手写链表多出额外的开销。对其他容器而言，size 保证是一个快速的常量时间的操作。

C++ 11

新标准库的容器比旧版本快得多，原因我们将在 13.6 节（第 470 页）解释。新标准库容器的性能几乎肯定与最精心优化过的同类数据结构一样好（通常会更好）。现代 C++程序应该使用标准库容器，而不是更原始的数据结构，如内置数组。

确定使用哪种顺序容器

通常，使用 vector 是最好的选择，除非你有很好的理由选择其他容器。

以下是一些选择容器的基本原则：

- 除非你有很好的理由选择其他容器，否则应使用 vector。
- 如果你的程序有很多小的元素，且空间的额外开销很重要，则不要使用 list 或 forward_list。
- 如果程序要求随机访问元素，应使用 vector 或 deque。
- 如果程序要求在容器的中间插入或删除元素，应使用 list 或 forward_list。
- 如果程序需要在头尾位置插入或删除元素，但不会在中间位置进行插入或删除操作，则使用 deque。
- 如果程序只有在读取输入时才需要在容器中间位置插入元素，随后需要随机访问元素，则 328
  - 首先，确定是否真的需要在容器中间位置添加元素。当处理输入数据时，通常可以很容易地向 vector 追加数据，然后再调用标准库的 sort 函数（我们将在 10.2.3 节介绍 sort（第 343 页））来重排容器中的元素，从而避免在中间位置添加元素。
  - 如果必须在中间位置插入元素，考虑在输入阶段使用 list，一旦输入完成，将 list 中的内容拷贝到一个 vector 中。

如果程序既需要随机访问元素，又需要在容器中间位置插入元素，那该怎么办？答案取决于在 list 或 forward_list 中访问元素与 vector 或 deque 中插入/删除元素的相对性能。一般来说，应用中占主导地位的操作（执行的访问操作更多还是插入/删除更多）决定了容器类型的选择。在此情况下，对两种容器分别测试应用的性能可能就是必要的了。

如果你不确定应该使用哪种容器，那么可以在程序中只使用 vector 和 list 公共的操作：使用迭代器，不使用下标操作，避免随机访问。这样，在必要时选择使用 vector 或 list 都很方便。

9.1 节练习

练习 9.1：对于下面的程序任务，vector、deque 和 list 哪种容器最为适合？解释你的选择的理由。如果没有哪一种容器优于其他容器，也请解释理由。

(a) 读取固定数量的单词，将它们按字典序插入到容器中。我们将在下一章中看到，关联容器更适合这个问题。

(b) 读取未知数量的单词，总是将新单词插入到末尾。删除操作在头部进行。

(c) 从一个文件读取未知数量的整数。将这些数排序，然后将它们打印到标准输出。

9.2 容器库概览

容器类型上的操作形成了一种层次：

- 某些操作是所有容器类型都提供的（参见表 9.2，第 295 页）。
- 另外一些操作仅针对顺序容器（参见表 9.3，第 299 页）、关联容器（参见表 11.7，第 388 页）或无序容器（参见表 11.8，第 395 页）。
- 还有一些操作只适用于一小部分容器。

329 在本节中，我们将介绍对所有容器都适用的操作。本章剩余部分将聚焦于仅适用于顺序容器的操作。关联容器特有的操作将在第 11 章介绍。

一般来说，每个容器都定义在一个头文件中，文件名与类型名相同。即，deque 定义在头文件 deque 中，list 定义在头文件 list 中，以此类推。容器均定义为模板类（参见 3.3 节，第 86 页）。例如对 vector，我们必须提供额外信息来生成特定的容器类型。对大多数，但不是所有容器，我们还需要额外提供元素类型信息：

```
list<Sales_data>      // 保存 Sales_data 对象的 list
deque<double>         // 保存 double 的 deque
```

对容器可以保存的元素类型的限制

顺序容器几乎可以保存任意类型的元素。特别是，我们可以定义一个容器，其元素的类型是另一个容器。这种容器的定义与任何其他容器类型完全一样：在尖括号中指定元素类型（此种情况下，是另一种容器类型）：

```
vector<vector<string>> lines; // vector 的 vector
```

C++ 11 此处 lines 是一个 vector，其元素类型是 string 的 vector。

较旧的编译器可能需要在两个尖括号之间键入空格，例如，vector<vector<string> >。

虽然我们可以在容器中保存几乎任何类型，但某些容器操作对元素类型有其自己的特殊要求。我们可以为不支持特定操作需求的类型定义容器，但这种情况下就只能使用那些没有特殊要求的容器操作了。

例如，顺序容器构造函数的一个版本接受容器大小参数（参见 3.3.1 节，第 88 页），它使用了元素类型的默认构造函数。但某些类没有默认构造函数。我们可以定义一个保存这种类型对象的容器，但我们在构造这种容器时不能只传递给它一个元素数目参数：

```
// 假定 noDefault 是一个没有默认构造函数的类型
vector<noDefault> v1(10, init);      // 正确：提供了元素初始化器
vector<noDefault> v2(10);            // 错误：必须提供一个元素初始化器
```

当后面介绍容器操作时，我们还会注意到每个容器操作对元素类型的其他限制。

330

表 9.2：容器操作

类型别名	
`iterator`	此容器类型的迭代器类型
`const_iterator`	可以读取元素，但不能修改元素的迭代器类型
`size_type`	无符号整数类型，足够保存此种容器类型最大可能容器的大小
`difference_type`	带符号整数类型，足够保存两个迭代器之间的距离
`value_type`	元素类型
`reference`	元素的左值类型；与 `value_type&` 含义相同
`const_reference`	元素的 const 左值类型（即，`const value_type&`）
构造函数	
`C c;`	默认构造函数，构造空容器（array，参见第 301 页）
`C c1(c2);`	构造 c2 的拷贝 c1
`C c(b, e);`	构造 c，将迭代器 b 和 e 指定的范围内的元素拷贝到 c（array 不支持）
`C c{a, b, c...};`	列表初始化 c
赋值与 swap	
`c1 = c2`	将 c1 中的元素替换为 c2 中元素
`c1 = {a, b, c...}`	将 c1 中的元素替换为列表中元素（不适用于 array）
`a.swap(b)`	交换 a 和 b 的元素
`swap(a, b)`	与 a.swap(b) 等价
大小	
`c.size()`	c 中元素的数目（不支持 forward_list）
`c.max_size()`	c 可保存的最大元素数目
`c.empty()`	若 c 中存储了元素，返回 false，否则返回 true
添加/删除元素（不适用于 array）	
注：在不同容器中，这些操作的接口都不同	
`c.insert(args)`	将 *args* 中的元素拷贝进 c
`c.emplace(inits)`	使用 *inits* 构造 c 中的一个元素
`c.erase(args)`	删除 *args* 指定的元素
`c.clear()`	删除 c 中的所有元素，返回 void
关系运算符	
`==, !=`	所有容器都支持相等（不等）运算符
`<, <=, >, >=`	关系运算符（无序关联容器不支持）
获取迭代器	
`c.begin(), c.end()`	返回指向 c 的首元素和尾元素之后位置的迭代器
`c.cbegin(), c.cend()`	返回 const_iterator

续表

反向容器的额外成员（不支持 forward_list）	
reverse_iterator	按逆序寻址元素的迭代器
const_reverse_iterator	不能修改元素的逆序迭代器
c.rbegin(), c.rend()	返回指向 c 的尾元素和首元素之前位置的迭代器
c.crbegin(), c.crend()	返回 const_reverse_iterator

9.2 节练习

练习 9.2：定义一个 list 对象，其元素类型是 int 的 deque。

331

9.2.1 迭代器

与容器一样，迭代器有着公共的接口：如果一个迭代器提供某个操作，那么所有提供相同操作的迭代器对这个操作的实现方式都是相同的。例如，标准容器类型上的所有迭代器都允许我们访问容器中的元素，而所有迭代器都是通过解引用运算符来实现这个操作的。类似的，标准库容器的所有迭代器都定义了递增运算符，从当前元素移动到下一个元素。

表 3.6（第 96 页）列出了容器迭代器支持的所有操作，其中有一个例外不符合公共接口特点——forward_list 迭代器不支持递减运算符（--）。表 3.7（第 99 页）列出了迭代器支持的算术运算，这些运算只能应用于 string、vector、deque 和 array 的迭代器。我们不能将它们用于其他任何容器类型的迭代器。

迭代器范围

迭代器范围的概念是标准库的基础。

一个**迭代器范围**（iterator range）由一对迭代器表示，两个迭代器分别指向同一个容器中的元素或者是尾元素之后的位置（one past the last element）。这两个迭代器通常被称为 begin 和 end，或者是 first 和 last（可能有些误导），它们标记了容器中元素的一个范围。

虽然第二个迭代器常常被称为 last，但这种叫法有些误导，因为第二个迭代器从来都不会指向范围中的最后一个元素，而是指向尾元素之后的位置。迭代器范围中的元素包含 first 所表示的元素以及从 first 开始直至 last（但不包含 last）之间的所有元素。

这种元素范围被称为**左闭合区间**（left-inclusive interval），其标准数学描述为

```
[begin, end)
```

表示范围自 begin 开始，于 end 之前结束。迭代器 begin 和 end 必须指向相同的容器。end 可以与 begin 指向相同的位置，但不能指向 begin 之前的位置。

对构成范围的迭代器的要求

如果满足如下条件，两个迭代器 begin 和 end 构成一个迭代器范：

- 它们指向同一个容器中的元素，或者是容器最后一个元素之后的位置，且
- 我们可以通过反复递增 begin 来到达 end。换句话说，end 不在 begin 之前。

编译器不会强制这些要求。确保程序符合这些约定是程序员的责任。

使用左闭合范围蕴含的编程假定

标准库使用左闭合范围是因为这种范围有三种方便的性质。假定 begin 和 end 构成一个合法的迭代器范围，则 332

- 如果 begin 与 end 相等，则范围为空
- 如果 begin 与 end 不等，则范围至少包含一个元素，且 begin 指向该范围中的第一个元素
- 我们可以对 begin 递增若干次，使得 begin==end

这些性质意味着我们可以像下面的代码一样用一个循环来处理一个元素范围，而这是安全的：

```
while (begin != end) {
    *begin = val;  // 正确：范围非空，因此 begin 指向一个元素
    ++begin;       // 移动迭代器，获取下一个元素
}
```

给定构成一个合法范围的迭代器 begin 和 end，若 begin==end，则范围为空。在此情况下，我们应该退出循环。如果范围不为空，begin 指向此非空范围的一个元素。因此，在 while 循环体中，可以安全地解引用 begin，因为 begin 必然指向一个元素。最后，由于每次循环对 begin 递增一次，我们确定循环最终会结束。

9.2.1 节练习

练习 9.3：构成迭代器范围的迭代器有何限制？

练习 9.4：编写函数，接受一对指向 vector<int>的迭代器和一个 int 值。在两个迭代器指定的范围中查找给定的值，返回一个布尔值来指出是否找到。

练习 9.5：重写上一题的函数，返回一个迭代器指向找到的元素。注意，程序必须处理未找到给定值的情况。

练习 9.6：下面程序有何错误？你应该如何修改它？

```
list<int> lst1;
list<int>::iterator iter1 = lst1.begin(),
                    iter2 = lst1.end();
while (iter1 < iter2) /* ... */
```

9.2.2 容器类型成员

每个容器都定义了多个类型，如表 9.2 所示（第 295 页）。我们已经使用过其中三种：size_type（参见 3.2.2 节，第 79 页）、iterator 和 const_iterator（参见 3.4.1 节，第 97 页）。

除了已经使用过的迭代器类型，大多数容器还提供反向迭代器。简单地说，反向迭代器就是一种反向遍历容器的迭代器，与正向迭代器相比，各种操作的含义也都发生了颠倒。例如，对一个反向迭代器执行++操作，会得到上一个元素。我们将在 10.4.3 节（第 363 页） 333

介绍更多关于反向迭代器的内容。

剩下的就是类型别名了，通过类型别名，我们可以在不了解容器中元素类型的情况下使用它。如果需要元素类型，可以使用容器的 value_type。如果需要元素类型的一个引用，可以使用 reference 或 const_reference。这些元素相关的类型别名在泛型编程中非常有用，我们将在 16 章中介绍相关内容。

为了使用这些类型，我们必须显式使用其类名：

```
// iter 是通过 list<string>定义的一个迭代器类型
list<string>::iterator iter;
// count 是通过 vector<int>定义的一个 difference_type 类型
vector<int>::difference_type count;
```

这些声明语句使用了作用域运算符（参见 1.2 节，第 7 页）来说明我们希望使用 list<string>类的 iterator 成员及 vector<int>类定义的 difference_type。

9.2.2 节练习

练习 9.7：为了索引 int 的 vector 中的元素，应该使用什么类型？

练习 9.8：为了读取 string 的 list 中的元素，应该使用什么类型？如果写入 list，又该使用什么类型？

9.2.3 begin 和 end 成员

begin 和 end 操作（参见 3.4.1 节，第 95 页）生成指向容器中第一个元素和尾元素之后位置的迭代器。这两个迭代器最常见的用途是形成一个包含容器中所有元素的迭代器范围。

如表 9.2（第 295 页）所示，begin 和 end 有多个版本：带 r 的版本返回反向迭代器（我们将在 10.4.3 节（第 363 页）中介绍相关内容）；以 c 开头的版本则返回 const 迭代器：

```
list<string> a = {"Milton", "Shakespeare", "Austen"};
auto it1 = a.begin();  // list<string>::iterator
auto it2 = a.rbegin(); // list<string>::reverse_iterator
auto it3 = a.cbegin(); // list<string>::const_iterator
auto it4 = a.crbegin();// list<string>::const_reverse_iterator
```

不以 c 开头的函数都是被重载过的。也就是说，实际上有两个名为 begin 的成员。一个是 const 成员（参见 7.1.2 节，第 231 页），返回容器的 const_iterator 类型。另一个是非常量成员，返回容器的 iterator 类型。rbegin、end 和 rend 的情况类似。当我们对一个非常量对象调用这些成员时，得到的是返回 iterator 的版本。只有在对一个 const 对象调用这些函数时，才会得到一个 const 版本。与 const 指针和引用类似，可以将一个普通的 iterator 转换为对应的 const_iterator，但反之不行。

334

以 c 开头的版本是 C++新标准引入的，用以支持 auto（参见 2.5.2 节，第 61 页）与 begin 和 end 函数结合使用。过去，没有其他选择，只能显式声明希望使用哪种类型的迭代器：

```
// 显式指定类型
list<string>::iterator it5 = a.begin();
```

```
list<string>::const_iterator it6 = a.begin();
// 是 iterator 还是 const_iterator 依赖于 a 的类型
auto it7 = a.begin();  // 仅当 a 是 const 时，it7 是 const_iterator
auto it8 = a.cbegin(); // it8 是 const_iterator
```

当 auto 与 begin 或 end 结合使用时，获得的迭代器类型依赖于容器类型，与我们想要如何使用迭代器毫不相干。但以 c 开头的版本还是可以获得 const_iterator 的，而不管容器的类型是什么。

当不需要写访问时，应使用 cbegin 和 cend。

9.2.3 节练习

练习 9.9：begin 和 cbegin 两个函数有什么不同？

练习 9.10：下面 4 个对象分别是什么类型？

```
vector<int> v1;
const vector<int> v2;
auto it1 = v1.begin(), it2 = v2.begin();
auto it3 = v1.cbegin(), it4 = v2.cbegin();
```

9.2.4 容器定义和初始化

每个容器类型都定义了一个默认构造函数（参见 7.1.4 节，第 236 页）。除 array 之外，其他容器的默认构造函数都会创建一个指定类型的空容器，且都可以接受指定容器大小和元素初始值的参数。

表 9.3：容器定义和初始化

C c;	默认构造函数。如果 C 是一个 array，则 c 中元素按默认方式初始化；否则 c 为空
C c1(c2) C c1=c2	c1 初始化为 c2 的拷贝。c1 和 c2 必须是相同类型（即，它们必须是相同的容器类型，且保存的是相同的元素类型；对于 array 类型，两者还必须具有相同大小）
C c{a,b,c...} C c={a,b,c...}	c 初始化为初始化列表中元素的拷贝。列表中元素的类型必须与 C 的元素类型相容。对于 array 类型，列表中元素数目必须等于或小于 array 的大小，任何遗漏的元素都进行值初始化（参见 3.3.1 节，第 88 页）
C c(b,e)	c 初始化为迭代器 b 和 e 指定范围中的元素的拷贝。范围中元素的类型必须与 C 的元素类型相容（array 不适用）
只有顺序容器（不包括 array）的构造函数才能接受大小参数	
C seq(n)	seq 包含 n 个元素，这些元素进行了值初始化；此构造函数是 explicit 的（参见 7.5.4 节，第 265 页）。（string 不适用）
C seq(n,t)	seq 包含 n 个初始化为值 t 的元素

将一个容器初始化为另一个容器的拷贝

将一个新容器创建为另一个容器的拷贝的方法有两种：可以直接拷贝整个容器，或者

(array 除外) 拷贝由一个迭代器对指定的元素范围。

为了创建一个容器为另一个容器的拷贝，两个容器的类型及其元素类型必须匹配。不过，当传递迭代器参数来拷贝一个范围时，就不要求容器类型是相同的了。而且，新容器和原容器中的元素类型也可以不同，只要能将要拷贝的元素转换（参见 4.11 节，第 141 页）为要初始化的容器的元素类型即可。

335

```
// 每个容器有三个元素，用给定的初始化器进行初始化
list<string> authors = {"Milton", "Shakespeare", "Austen"};
vector<const char*> articles = {"a", "an", "the"};

list<string> list2(authors);      // 正确：类型匹配
deque<string> authList(authors); // 错误：容器类型不匹配
vector<string> words(articles);  // 错误：容器类型必须匹配
// 正确：可以将 const char*元素转换为 string
forward_list<string> words(articles.begin(), articles.end());
```

当将一个容器初始化为另一个容器的拷贝时，两个容器的容器类型和元素类型都必须相同。

接受两个迭代器参数的构造函数用这两个迭代器表示我们想要拷贝的一个元素范围。与以往一样，两个迭代器分别标记想要拷贝的第一个元素和尾元素之后的位置。新容器的大小与范围中元素的数目相同。新容器中的每个元素都用范围中对应元素的值进行初始化。

由于两个迭代器表示一个范围，因此可以使用这种构造函数来拷贝一个容器中的子序列。例如，假定迭代器 it 表示 authors 中的一个元素，我们可以编写如下代码

```
// 拷贝元素，直到（但不包括）it 指向的元素
deque<string> authList(authors.begin(), it);
```

336

列表初始化

C++ 11

在新标准中，我们可以对一个容器进行列表初始化（参见 3.3.1 节，第 88 页）

```
// 每个容器有三个元素，用给定的初始化器进行初始化
list<string> authors = {"Milton", "Shakespeare", "Austen"};
vector<const char*> articles = {"a", "an", "the"};
```

当这样做时，我们就显式地指定了容器中每个元素的值。对于除 array 之外的容器类型，初始化列表还隐含地指定了容器的大小：容器将包含与初始值一样多的元素。

与顺序容器大小相关的构造函数

除了与关联容器相同的构造函数外，顺序容器（array 除外）还提供另一个构造函数，它接受一个容器大小和一个（可选的）元素初始值。如果我们不提供元素初始值，则标准库会创建一个值初始化器（参见 3.3.1 节，第 88 页）：

```
vector<int> ivec(10, -1);          // 10 个 int 元素，每个都初始化为-1
list<string> svec(10, "hi!");      // 10 个 strings；每个都初始化为"hi!"
forward_list<int> ivec(10);        // 10 个元素，每个都初始化为 0
deque<string> svec(10);            // 10 个元素，每个都是空 string
```

如果元素类型是内置类型或者是具有默认构造函数（参见 9.2 节，第 294 页）的类类型，可以只为构造函数提供一个容器大小参数。如果元素类型没有默认构造函数，除了大小参数外，还必须指定一个显式的元素初始值。

只有顺序容器的构造函数才接受大小参数，关联容器并不支持。

标准库 array 具有固定大小

与内置数组一样，标准库 array 的大小也是类型的一部分。当定义一个 array 时，除了指定元素类型，还要指定容器大小：

```
array<int, 42>       // 类型为：保存 42 个 int 的数组
array<string, 10>    // 类型为：保存 10 个 string 的数组
```

为了使用 array 类型，我们必须同时指定元素类型和大小：

```
array<int, 10>::size_type i;      // 数组类型包括元素类型和大小
array<int>::size_type j;          // 错误：array<int>不是一个类型
```

由于大小是 array 类型的一部分，array 不支持普通的容器构造函数。这些构造函数都会确定容器的大小，要么隐式地，要么显式地。而允许用户向一个 array 构造函数传递大小参数，最好情况下也是多余的，而且容易出错。

array 大小固定的特性也影响了它所定义的构造函数的行为。与其他容器不同，一个默认构造的 array 是非空的：它包含了与其大小一样多的元素。这些元素都被默认初始化（参见 2.2.1 节，第 40 页），就像一个内置数组（参见 3.5.1 节，第 102 页）中的元素那样。如果我们对 array 进行列表初始化，初始值的数目必须等于或小于 array 的大小。如果初始值数目小于 array 的大小，则它们被用来初始化 array 中靠前的元素，所有剩余元素都会进行值初始化（参见 3.3.1 节，第 88 页）。在这两种情况下，如果元素类型是一个类类型，那么该类必须有一个默认构造函数，以使值初始化能够进行：

337

```
array<int, 10> ia1;           // 10 个默认初始化的 int
array<int, 10> ia2 = {0,1,2,3,4,5,6,7,8,9}; // 列表初始化
array<int, 10> ia3 = {42};    // ia3[0]为 42，剩余元素为 0
```

值得注意的是，虽然我们不能对内置数组类型进行拷贝或对象赋值操作（参见 3.5.1 节，第 102 页），但 array 并无此限制：

```
int digs[10] = {0,1,2,3,4,5,6,7,8,9};
int cpy[10] = digs;              // 错误：内置数组不支持拷贝或赋值
array<int, 10> digits = {0,1,2,3,4,5,6,7,8,9};
array<int, 10> copy = digits; // 正确：只要数组类型匹配即合法
```

与其他容器一样，array 也要求初始值的类型必须与要创建的容器类型相同。此外，array 还要求元素类型和大小也都一样，因为大小是 array 类型的一部分。

9.2.4 节练习

练习 9.11：对 6 种创建和初始化 vector 对象的方法，每一种都给出一个实例。解释每个 vector 包含什么值。

练习 9.12：对于接受一个容器创建其拷贝的构造函数，和接受两个迭代器创建拷贝的构造函数，解释它们的不同。

练习 9.13：如何从一个 list<int>初始化一个 vector<double>？从一个 vector<int>又该如何创建？编写代码验证你的答案。

9.2.5 赋值和 swap

表 9.4 中列出的与赋值相关的运算符可用于所有容器。赋值运算符将其左边容器中的全部元素替换为右边容器中元素的拷贝：

```
c1 = c2;        // 将 c1 的内容替换为 c2 中元素的拷贝
c1 = {a,b,c};   // 赋值后，c1 大小为 3
```

第一个赋值运算后，左边容器将与右边容器相等。如果两个容器原来大小不同，赋值运算后两者的大小都与右边容器的原大小相同。第二个赋值运算后，c1 的 size 变为 3，即花括号列表中值的数目。

338 与内置数组不同，标准库 array 类型允许赋值。赋值号左右两边的运算对象必须具有相同的类型：

```
array<int, 10> a1 = {0,1,2,3,4,5,6,7,8,9};
array<int, 10> a2 = {0}; // 所有元素值均为 0
a1 = a2;  // 替换 a1 中的元素
a2 = {0}; // 错误：不能将一个花括号列表赋予数组
```

由于右边运算对象的大小可能与左边运算对象的大小不同，因此 array 类型不支持 assign，也不允许用花括号包围的值列表进行赋值。

表 9.4：容器赋值运算

c1=c2	将 c1 中的元素替换为 c2 中元素的拷贝。c1 和 c2 必须具有相同的类型
c={a,b,c...}	将 c1 中元素替换为初始化列表中元素的拷贝（array 不适用）
swap(c1,c2) c1.swap(c2)	交换 c1 和 c2 中的元素。c1 和 c2 必须具有相同的类型。swap 通常比从 c2 向 c1 拷贝元素快得多
assign 操作不适用于关联容器和 array	
seq.assign(b,e)	将 seq 中的元素替换为迭代器 b 和 e 所表示的范围中的元素。迭代器 b 和 e 不能指向 seq 中的元素
seq.assign(il)	将 seq 中的元素替换为初始化列表 il 中的元素
seq.assign(n,t)	将 seq 中的元素替换为 n 个值为 t 的元素

赋值相关运算会导致指向左边容器内部的迭代器、引用和指针失效。而 swap 操作将容器内容交换不会导致指向容器的迭代器、引用和指针失效（容器类型为 array 和 string 的情况除外）。

使用 assign（仅顺序容器）

赋值运算符要求左边和右边的运算对象具有相同的类型。它将右边运算对象中所有元素拷贝到左边运算对象中。顺序容器（array 除外）还定义了一个名为 assign 的成员，允许我们从一个不同但相容的类型赋值，或者从容器的一个子序列赋值。assign 操作用参数所指定的元素（的拷贝）替换左边容器中的所有元素。例如，我们可以用 assgin 实现将一个 vector 中的一段 char *值赋予一个 list 中的 string：

```
list<string> names;
vector<const char*> oldstyle;
names = oldstyle; // 错误：容器类型不匹配
// 正确：可以将 const char*转换为 string
names.assign(oldstyle.cbegin(), oldstyle.cend());
```

这段代码中对 assign 的调用将 names 中的元素替换为迭代器指定的范围中的元素的拷贝。assign 的参数决定了容器中将有多少个元素以及它们的值都是什么。 339

由于其旧元素被替换，因此传递给 assign 的迭代器不能指向调用 assign 的容器。

assign 的第二个版本接受一个整型值和一个元素值。它用指定数目且具有相同给定值的元素替换容器中原有的元素：

```
// 等价于 slist1.clear();
// 后跟 slist1.insert(slist1.begin(), 10, "Hiya!");
list<string> slist1(1);       // 1 个元素，为空 string
slist1.assign(10, "Hiya!");  // 10 个元素，每个都是“Hiya!”
```

使用 swap

swap 操作交换两个相同类型容器的内容。调用 swap 之后，两个容器中的元素将会交换：

```
vector<string> svec1(10); // 10 个元素的 vector
vector<string> svec2(24); // 24 个元素的 vector
swap(svec1, svec2);
```

调用 swap 后，svec1 将包含 24 个 string 元素，svec2 将包含 10 个 string。除 array 外，交换两个容器内容的操作保证会很快——元素本身并未交换，swap 只是交换了两个容器的内部数据结构。

除 array 外，swap 不对任何元素进行拷贝、删除或插入操作，因此可以保证在常数时间内完成。

元素不会被移动的事实意味着，除 string 外，指向容器的迭代器、引用和指针在 swap 操作之后都不会失效。它们仍指向 swap 操作之前所指向的那些元素。但是，在 swap 之后，这些元素已经属于不同的容器了。例如，假定 iter 在 swap 之前指向 svec1[3] 的 string，那么在 swap 之后它指向 svec2[3] 的元素。与其他容器不同，对一个 string 调用 swap 会导致迭代器、引用和指针失效。

与其他容器不同，swap 两个 array 会真正交换它们的元素。因此，交换两个 array 所需的时间与 array 中元素的数目成正比。

因此，对于 array，在 swap 操作之后，指针、引用和迭代器所绑定的元素保持不变，但元素值已经与另一个 array 中对应元素的值进行了交换。

在新标准库中，容器既提供成员函数版本的 swap，也提供非成员版本的 swap。而早期标准库版本只提供成员函数版本的 swap。非成员版本的 swap 在泛型编程中是非常重要的。统一使用非成员版本的 swap 是一个好习惯。 C++ 11

340

9.2.5 节练习

练习 9.14：编写程序，将一个 list 中的 char *指针（指向 C 风格字符串）元素赋值给一个 vector 中的 string。

9.2.6 容器大小操作

除了一个例外，每个容器类型都有三个与大小相关的操作。成员函数 size（参见 3.2.2 节，第 78 页）返回容器中元素的数目；empty 当 size 为 0 时返回布尔值 true，否则返回 false；max_size 返回一个大于或等于该类型容器所能容纳的最大元素数的值。forward_list 支持 max_size 和 empty，但不支持 size，原因我们将在下一节解释。

9.2.7 关系运算符

每个容器类型都支持相等运算符（==和!=）；除了无序关联容器外的所有容器都支持关系运算符（>、>=、<、<=）。关系运算符左右两边的运算对象必须是相同类型的容器，且必须保存相同类型的元素。即，我们只能将一个 vector<int>与另一个 vector<int>进行比较，而不能将一个 vector<int>与一个 list<int>或一个 vector<double>进行比较。

比较两个容器实际上是进行元素的逐对比较。这些运算符的工作方式与 string 的关系运算（参见 3.2.2 节，第 79 页）类似：

- 如果两个容器具有相同大小且所有元素都两两对应相等，则这两个容器相等；否则两个容器不等。
- 如果两个容器大小不同，但较小容器中每个元素都等于较大容器中的对应元素，则较小容器小于较大容器。
- 如果两个容器都不是另一个容器的前缀子序列，则它们的比较结果取决于第一个不相等的元素的比较结果。

下面的例子展示了这些关系运算符是如何工作的：

```
vector<int> v1 = { 1, 3, 5, 7, 9, 12 };
vector<int> v2 = { 1, 3, 9 };
vector<int> v3 = { 1, 3, 5, 7 };
vector<int> v4 = { 1, 3, 5, 7, 9, 12 };
v1 < v2  // true; v1 和 v2 在元素[2]处不同：v1[2]小于等于 v2[2]
v1 < v3  // false; 所有元素都相等，但 v3 中元素数目更少
v1 == v4 // true; 每个元素都相等，且 v1 和 v4 大小相同
v1 == v2 // false; v2 元素数目比 v1 少
```

341

容器的关系运算符使用元素的关系运算符完成比较

只有当其元素类型也定义了相应的比较运算符时，我们才可以使用关系运算符来比较两个容器。

容器的相等运算符实际上是使用元素的==运算符实现比较的，而其他关系运算符是使用元素的<运算符。如果元素类型不支持所需运算符，那么保存这种元素的容器就不能使用相应的关系运算。例如，我们在第 7 章中定义的 Sales_data 类型并未定义==和<运算。因此，就不能比较两个保存 Sales_data 元素的容器：

```
vector<Sales_data> storeA, storeB;
if (storeA < storeB) // 错误：Sales_data 没有<运算符
```

9.2.7 节练习

练习 9.15：编写程序，判定两个 vector<int>是否相等。

练习 9.16：重写上一题的程序，比较一个 list<int>中的元素和一个 vector<int>中的元素。

练习 9.17：假定 c1 和 c2 是两个容器，下面的比较操作有何限制（如果有的话）？

```
if (c1 < c2)
```

9.3 顺序容器操作

顺序容器和关联容器的不同之处在于两者组织元素的方式。这些不同之处直接关系到了元素如何存储、访问、添加以及删除。上一节介绍了所有容器都支持的操作（罗列于表 9.2（第 295 页））。本章剩余部分将介绍顺序容器所特有的操作。

9.3.1 向顺序容器添加元素

除 array 外，所有标准库容器都提供灵活的内存管理。在运行时可以动态添加或删除元素来改变容器大小。表 9.5 列出了向顺序容器（非 array）添加元素的操作。

表 9.5：向顺序容器添加元素的操作

这些操作会改变容器的大小；array 不支持这些操作。
forward_list 有自己专有版本的 insert 和 emplace；参见 9.3.4 节（第 312 页）。
forward_list 不支持 push_back 和 emplace_back。
vector 和 string 不支持 push_front 和 emplace_front。

c.push_back(t) c.emplace_back(*args*)	在 c 的尾部创建一个值为 t 或由 *args* 创建的元素。返回 void
c.push_front(t) c.emplace_front(*args*)	在 c 的头部创建一个值为 t 或由 *args* 创建的元素。返回 void
c.insert(p,t) c.emplace(p,*args*)	在迭代器 p 指向的元素之前创建一个值为 t 或由 *args* 创建的元素。返回指向新添加的元素的迭代器
c.insert(p,n,t)	在迭代器 p 指向的元素之前插入 n 个值为 t 的元素。返回指向新添加的第一个元素的迭代器；若 n 为 0，则返回 p
c.insert(p,b,e)	将迭代器 b 和 e 指定的范围内的元素插入到迭代器 p 指向的元素之前。b 和 e 不能指向 c 中的元素。返回指向新添加的第一个元素的迭代器；若范围为空，则返回 p
c.insert(p,il)	il 是一个花括号包围的元素值列表。将这些给定值插入到迭代器 p 指向的元素之前。返回指向新添加的第一个元素的迭代器；若列表为空，则返回 p

向一个 vector、string 或 deque 插入元素会使所有指向容器的迭代器、引用和指针失效。

当我们使用这些操作时，必须记得不同容器使用不同的策略来分配元素空间，而这些策略直接影响性能。在一个 vector 或 string 的尾部之外的任何位置，或是一个 deque 的首尾之外的任何位置添加元素，都需要移动元素。而且，向一个 vector 或 string 添
342 加元素可能引起整个对象存储空间的重新分配。重新分配一个对象的存储空间需要分配新的内存，并将元素从旧的空间移动到新的空间中。

使用 push_back

在 3.3.2 节（第 90 页）中，我们看到 push_back 将一个元素追加到一个 vector 的尾部。除 array 和 forward_list 之外，每个顺序容器（包括 string 类型）都支持 push_back。

例如，下面的循环每次读取一个 string 到 word 中，然后追加到容器尾部：

```
// 从标准输入读取数据，将每个单词放到容器末尾
string word;
while (cin >> word)
    container.push_back(word);
```

对 push_back 的调用在 container 尾部创建了一个新的元素，将 container 的 size 增大了 1。该元素的值为 word 的一个拷贝。container 的类型可以是 list、vector 或 deque。

由于 string 是一个字符容器，我们也可以用 push_back 在 string 末尾添加字符：

```
void pluralize(size_t cnt, string &word)
{
    if (cnt > 1)
        word.push_back('s'); // 等价于 word += 's'
}
```

关键概念：容器元素是拷贝

当我们用一个对象来初始化容器时，或将一个对象插入到容器中时，实际上放入到容器中的是对象值的一个拷贝，而不是对象本身。就像我们将一个对象传递给非引用参数（参见 3.2.2 节，第 79 页）一样，容器中的元素与提供值的对象之间没有任何关联。随后对容器中元素的任何改变都不会影响到原始对象，反之亦然。

使用 push_front

除了 push_back，list、forward_list 和 deque 容器还支持名为 push_front 的类似操作。此操作将元素插入到容器头部：

```
list<int> ilist;
// 将元素添加到 ilist 开头
for (size_t ix = 0; ix != 4; ++ix)
    ilist.push_front(ix);
```

此循环将元素 0、1、2、3 添加到 ilist 头部。每个元素都插入到 list 的新的开始位
343 置（new beginning）。即，当我们插入 1 时，它会被放置在 0 之前，2 被放置在 1 之前，依此类推。因此，在循环中以这种方式将元素添加到容器中，最终会形成逆序。在循环执行完毕后，ilist 保存序列 3、2、1、0。

注意，deque 像 vector 一样提供了随机访问元素的能力，但它提供了 vector 所

不支持的 push_front。deque 保证在容器首尾进行插入和删除元素的操作都只花费常数时间。与 vector 一样，在 deque 首尾之外的位置插入元素会很耗时。

在容器中的特定位置添加元素

push_back和push_front操作提供了一种方便地在顺序容器尾部或头部插入单个元素的方法。insert 成员提供了更一般的添加功能，它允许我们在容器中任意位置插入0个或多个元素。vector、deque、list和string都支持insert成员。forward_list提供了特殊版本的 insert 成员，我们将在 9.3.4 节（第 312 页）中介绍。

每个 insert 函数都接受一个迭代器作为其第一个参数。迭代器指出了在容器中什么位置放置新元素。它可以指向容器中任何位置，包括容器尾部之后的下一个位置。由于迭代器可能指向容器尾部之后不存在的元素的位置，而且在容器开始位置插入元素是很有用的功能，所以 insert 函数将元素插入到迭代器所指定的位置之前。例如，下面的语句

344

```
slist.insert(iter, "Hello!"); // 将"Hello!"添加到 iter 之前的位置
```

将一个值为"Hello"的 string 插入到 iter 指向的元素之前的位置。

虽然某些容器不支持 push_front 操作，但它们对于 insert 操作并无类似的限制（插入开始位置）。因此我们可以将元素插入到容器的开始位置，而不必担心容器是否支持 push_front：

```
vector<string> svec;
list<string> slist;

// 等价于调用 slist.push_front("Hello!");
slist.insert(slist.begin(), "Hello!");

// vector 不支持 push_front，但我们可以插入到 begin()之前
// 警告：插入到 vector 末尾之外的任何位置都可能很慢
svec.insert(svec.begin(), "Hello!");
```

将元素插入到 vector、deque 和 string 中的任何位置都是合法的。然而，这样做可能很耗时。

插入范围内元素

除了第一个迭代器参数之外，insert 函数还可以接受更多的参数，这与容器构造函数类似。其中一个版本接受一个元素数目和一个值，它将指定数量的元素添加到指定位置之前，这些元素都按给定值初始化：

```
svec.insert(svec.end(), 10, "Anna");
```

这行代码将 10 个元素插入到 svec 的末尾，并将所有元素都初始化为 string "Anna"。

接受一对迭代器或一个初始化列表的 insert 版本将给定范围中的元素插入到指定位置之前：

```
vector<string> v = {"quasi", "simba", "frollo", "scar"};
// 将 v 的最后两个元素添加到 slist 的开始位置
slist.insert(slist.begin(), v.end() - 2, v.end());
slist.insert(slist.end(), {"these", "words", "will",
                           "go", "at", "the", "end"});
```

```
// 运行时错误：迭代器表示要拷贝的范围，不能指向与目的位置相同的容器
slist.insert(slist.begin(), slist.begin(), slist.end());
```

如果我们传递给 insert 一对迭代器，它们不能指向添加元素的目标容器。

C++11 在新标准下，接受元素个数或范围的 insert 版本返回指向第一个新加入元素的迭代器。(在旧版本的标准库中，这些操作返回 void。)如果范围为空，不插入任何元素，insert 操作会将第一个参数返回。

345 ### 使用 insert 的返回值

通过使用 insert 的返回值，可以在容器中一个特定位置反复插入元素：

```
list<string> lst;
auto iter = lst.begin();
while (cin >> word)
    iter = lst.insert(iter, word); // 等价于调用 push_front
```

理解这个循环是如何工作的非常重要，特别是理解这个循环为什么等价于调用 push_front 尤为重要。

在循环之前，我们将 iter 初始化为 lst.begin()。第一次调用 insert 会将我们刚刚读入的 string 插入到 iter 所指向的元素之前的位置。insert 返回的迭代器恰好指向这个新元素。我们将此迭代器赋予 iter 并重复循环，读取下一个单词。只要继续有单词读入，每步 while 循环就会将一个新元素插入到 iter 之前，并将 iter 改变为新加入元素的位置。此元素为（新的）首元素。因此，每步循环将一个新元素插入到 list 首元素之前的位置。

使用 emplace 操作

C++11 新标准引入了三个新成员——emplace_front、emplace 和 emplace_back，这些操作构造而不是拷贝元素。这些操作分别对应 push_front、insert 和 push_back，允许我们将元素放置在容器头部、一个指定位置之前或容器尾部。

当调用 push 或 insert 成员函数时，我们将元素类型的对象传递给它们，这些对象被拷贝到容器中。而当我们调用一个 emplace 成员函数时，则是将参数传递给元素类型的构造函数。emplace 成员使用这些参数在容器管理的内存空间中直接构造元素。例如，假定 c 保存 Sales_data（参见 7.1.4 节，第 237 页）元素：

```
// 在 c 的末尾构造一个 Sales_data 对象
// 使用三个参数的 Sales_data 构造函数
c.emplace_back("978-0590353403", 25, 15.99);
// 错误：没有接受三个参数的 push_back 版本
c.push_back("978-0590353403", 25, 15.99);
// 正确：创建一个临时的 Sales_data 对象传递给 push_back
c.push_back(Sales_data("978-0590353403", 25, 15.99));
```

其中对 emplace_back 的调用和第二个 push_back 调用都会创建新的 Sales_data 对象。在调用 emplace_back 时，会在容器管理的内存空间中直接创建对象。而调用 push_back 则会创建一个局部临时对象，并将其压入容器中。

emplace 函数的参数根据元素类型而变化，参数必须与元素类型的构造函数相匹配：

346
```
// iter 指向 c 中一个元素，其中保存了 Sales_data 元素
```

```
c.emplace_back(); // 使用 Sales_data 的默认构造函数
c.emplace(iter, "999-999999999"); // 使用 Sales_data(string)
// 使用 Sales_data 的接受一个 ISBN、一个 count 和一个 price 的构造函数
c.emplace_front("978-0590353403", 25, 15.99);
```

emplace 函数在容器中直接构造元素。传递给 emplace 函数的参数必须与元素类型的构造函数相匹配。

9.3.1 节练习

练习 9.18：编写程序，从标准输入读取 string 序列，存入一个 deque 中。编写一个循环，用迭代器打印 deque 中的元素。

练习 9.19：重写上题的程序，用 list 替代 deque。列出程序要做出哪些改变。

练习 9.20：编写程序，从一个 list<int>拷贝元素到两个 deque 中。值为偶数的所有元素都拷贝到一个 deque 中，而奇数值元素都拷贝到另一个 deque 中。

练习 9.21：如果我们将第 308 页中使用 insert 返回值将元素添加到 list 中的循环程序改写为将元素插入到 vector 中，分析循环将如何工作。

练习 9.22：假定 iv 是一个 int 的 vector，下面的程序存在什么错误？你将如何修改？

```
vector<int>::iterator iter = iv.begin(),
                      mid = iv.begin() + iv.size()/2;
while (iter != mid)
     if (*iter == some_val)
          iv.insert(iter, 2 * some_val);
```

9.3.2 访问元素

表 9.6 列出了我们可以用来在顺序容器中访问元素的操作。如果容器中没有元素，访问操作的结果是未定义的。

包括 array 在内的每个顺序容器都有一个 front 成员函数，而除 forward_list 之外的所有顺序容器都有一个 back 成员函数。这两个操作分别返回首元素和尾元素的引用：

```
// 在解引用一个迭代器或调用 front 或 back 之前检查是否有元素
if (!c.empty()) {
     // val 和 val2 是 c 中第一个元素值的拷贝
     auto val = *c.begin(), val2 = c.front();
     // val3 和 val4 是 c 中最后一个元素值的拷贝
     auto last = c.end();
     auto val3 = *(--last); // 不能递减 forward_list 迭代器
     auto val4 = c.back();  // forward_list 不支持
}
```

此程序用两种不同方式来获取 c 中的首元素和尾元素的引用。直接的方法是调用 front 和 back。而间接的方法是通过解引用 begin 返回的迭代器来获得首元素的引用，以及通过递减然后解引用 end 返回的迭代器来获得尾元素的引用。 347

这个程序有两点值得注意：迭代器 end 指向的是容器尾元素之后的（不存在的）元

素。为了获取尾元素，必须首先递减此迭代器。另一个重要之处是，在调用 front 和 back（或解引用 begin 和 end 返回的迭代器）之前，要确保 c 非空。如果容器为空，if 中操作的行为将是未定义的。

表 9.6：在顺序容器中访问元素的操作

at 和下标操作只适用于 string、vector、deque 和 array。
back 不适用于 forward_list。

c.back()	返回 c 中尾元素的引用。若 c 为空，函数行为未定义
c.front()	返回 c 中首元素的引用。若 c 为空，函数行为未定义
c[n]	返回 c 中下标为 n 的元素的引用，n 是一个无符号整数。若 n>=c.size()，则函数行为未定义
c.at(n)	返回下标为 n 的元素的引用。如果下标越界，则抛出一 out_of_range 异常

对一个空容器调用 front 和 back，就像使用一个越界的下标一样，是一种严重的程序设计错误。

访问成员函数返回的是引用

在容器中访问元素的成员函数（即，front、back、下标和 at）返回的都是引用。如果容器是一个 const 对象，则返回值是 const 的引用。如果容器不是 const 的，则返回值是普通引用，我们可以用来改变元素的值：

```
if (!c.empty()) {
    c.front() = 42;          // 将 42 赋予 c 中的第一个元素
    auto &v = c.back();      // 获得指向最后一个元素的引用
    v = 1024;                // 改变 c 中的元素
    auto v2 = c.back();      // v2 不是一个引用，它是 c.back()的一个拷贝
    v2 = 0;                  // 未改变 c 中的元素
}
```

与往常一样，如果我们使用 auto 变量来保存这些函数的返回值，并且希望使用此变量来改变元素的值，必须记得将变量定义为引用类型。

下标操作和安全的随机访问

提供快速随机访问的容器（string、vector、deque 和 array）也都提供下标运
348
算符（参见 3.3.3 节，第 91 页）。就像我们已经看到的那样，下标运算符接受一个下标参数，返回容器中该位置的元素的引用。给定下标必须“在范围内”（即，大于等于 0，且小于容器的大小）。保证下标有效是程序员的责任，下标运算符并不检查下标是否在合法范围内。使用越界的下标是一种严重的程序设计错误，而且编译器并不检查这种错误。

如果我们希望确保下标是合法的，可以使用 at 成员函数。at 成员函数类似下标运算符，但如果下标越界，at 会抛出一个 out_of_range 异常（参见 5.6 节，第 173 页）：

```
vector<string> svec;      // 空 vector
cout << svec[0];          // 运行时错误：svec 中没有元素！
cout << svec.at(0);       // 抛出一个 out_of_range 异常
```

9.3.2 节练习

练习 9.23：在本节第一个程序（第 309 页）中，若 c.size() 为 1，则 val、val2、val3 和 val4 的值会是什么？

练习 9.24：编写程序，分别使用 at、下标运算符、front 和 begin 提取一个 vector 中的第一个元素。在一个空 vector 上测试你的程序。

9.3.3 删除元素

与添加元素的多种方式类似，（非 array）容器也有多种删除元素的方式。表 9.7 列出了这些成员函数。

表 9.7：顺序容器的删除操作

这些操作会改变容器的大小，所以不适用于 array。
forward_list 有特殊版本的 erase，参见 9.3.4 节（第 312 页）。
forward_list 不支持 pop_back；vector 和 string 不支持 pop_front。

c.pop_back()	删除 c 中尾元素。若 c 为空，则函数行为未定义。函数返回 void
c.pop_front()	删除 c 中首元素。若 c 为空，则函数行为未定义。函数返回 void
c.erase(p)	删除迭代器 p 所指定的元素，返回一个指向被删元素之后元素的迭代器，若 p 指向尾元素，则返回尾后（off-the-end）迭代器。若 p 是尾后迭代器，则函数行为未定义
c.erase(b,e)	删除迭代器 b 和 e 所指定范围内的元素。返回一个指向最后一个被删元素之后元素的迭代器，若 e 本身就是尾后迭代器，则函数也返回尾后迭代器
c.clear()	删除 c 中的所有元素。返回 void

删除 deque 中除首尾位置之外的任何元素都会使所有迭代器、引用和指针失效。指向 vector 或 string 中删除点之后位置的迭代器、引用和指针都会失效。

删除元素的成员函数并不检查其参数。在删除元素之前，程序员必须确保它（们）是存在的。

pop_front 和 pop_back 成员函数

pop_front 和 pop_back 成员函数分别删除首元素和尾元素。与 vector 和 string 不支持 push_front 一样，这些类型也不支持 pop_front。类似的，forward_list 不支持 pop_back。与元素访问成员函数类似，不能对一个空容器执行弹出操作。

这些操作返回 void。如果你需要弹出的元素的值，就必须在执行弹出操作之前保存它：

```
while (!ilist.empty()) {
    process(ilist.front()); // 对 ilist 的首元素进行一些处理
    ilist.pop_front();      // 完成处理后删除首元素
}
```

349 从容器内部删除一个元素

成员函数 erase 从容器中指定位置删除元素。我们可以删除由一个迭代器指定的单个元素，也可以删除由一对迭代器指定的范围内的所有元素。两种形式的 erase 都返回指向删除的(最后一个)元素之后位置的迭代器。即，若 j 是 i 之后的元素，那么 erase(i) 将返回指向 j 的迭代器。

例如，下面的循环删除一个 list 中的所有奇数元素：

```
list<int> lst = {0,1,2,3,4,5,6,7,8,9};
auto it = lst.begin();
while (it != lst.end())
    if (*it % 2)              // 若元素为奇数
        it = lst.erase(it);  // 删除此元素
    else
        ++it;
```

每个循环步中，首先检查当前元素是否是奇数。如果是，就删除该元素，并将 it 设置为我们所删除的元素之后的元素。如果*it 为偶数，我们将 it 递增，从而在下一步循环检查下一个元素。

删除多个元素

接受一对迭代器的 erase 版本允许我们删除一个范围内的元素：

```
// 删除两个迭代器表示的范围内的元素
// 返回指向最后一个被删元素之后位置的迭代器
elem1 = slist.erase(elem1, elem2); // 调用后，elem1 == elem2
```

迭代器 elem1 指向我们要删除的第一个元素，elem2 指向我们要删除的最后一个元素之后的位置。

350

为了删除一个容器中的所有元素，我们既可以调用 clear，也可以用 begin 和 end 获得的迭代器作为参数调用 erase：

```
slist.clear(); // 删除容器中所有元素
slist.erase(slist.begin(), slist.end()); // 等价调用
```

9.3.3 节练习

练习 9.25：对于第 312 页中删除一个范围内的元素的程序，如果 elem1 与 elem2 相等会发生什么？如果 elem2 是尾后迭代器，或者 elem1 和 elem2 皆为尾后迭代器，又会发生什么？

练习 9.26：使用下面代码定义的 ia，将 ia 拷贝到一个 vector 和一个 list 中。使用单迭代器版本的 erase 从 list 中删除奇数元素，从 vector 中删除偶数元素。

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
```

9.3.4 特殊的 forward_list 操作

为了理解 forward_list 为什么有特殊版本的添加和删除操作，考虑当我们从一个单向链表中删除一个元素时会发生什么。如图 9.1 所示，删除一个元素会改变序列中的链接。在此情况下，删除 $elem_3$ 会改变 $elem_2$，$elem_2$ 原来指向 $elem_3$，但删除 $elem_3$ 后，$elem_2$ 指向了 $elem_4$。

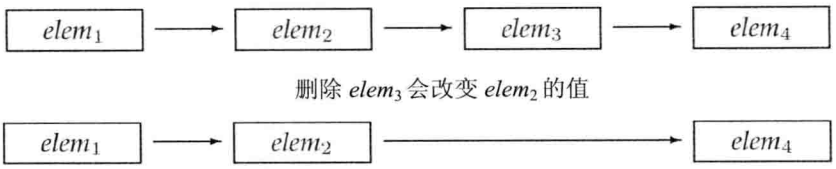

图 9.1：forward_list 的特殊操作

当添加或删除一个元素时，删除或添加的元素之前的那个元素的后继会发生改变。为了添加或删除一个元素，我们需要访问其前驱，以便改变前驱的链接。但是，forward_list 是单向链表。在一个单向链表中，没有简单的方法来获取一个元素的前驱。出于这个原因，在一个 forward_list 中添加或删除元素的操作是通过改变给定元素之后的元素来完成的。这样，我们总是可以访问到被添加或删除操作所影响的元素。

由于这些操作与其他容器上的操作的实现方式不同，forward_list 并未定义 insert、emplace 和 erase，而是定义了名为 insert_after、emplace_after 和 erase_after 的操作（参见表 9.8）。例如，在我们的例子中，为了删除 $elem_3$，应该用指向 $elem_2$ 的迭代器调用 erase_after。为了支持这些操作，forward_list 也定义了 before_begin，它返回一个**首前**（off-the-beginning）迭代器。这个迭代器允许我们在链表首元素之前并不存在的元素“之后”添加或删除元素（亦即在链表首元素之前添加删除元素）。 351

表 9.8：在 forward_list 中插入或删除元素的操作

lst.before_begin() lst.cbefore_begin()	返回指向链表首元素之前不存在的元素的迭代器。此迭代器不能解引用。cbefore_begin()返回一个 const_iterator
lst.insert_after(p,t) lst.insert_after(p,n,t) lst.insert_after(p,b,e) lst.insert_after(p,il)	在迭代器 p 之后的位置插入元素。t 是一个对象，n 是数量，b 和 e 是表示范围的一对迭代器（b 和 e 不能指向 lst 内），il 是一个花括号列表。返回一个指向最后一个插入元素的迭代器。如果范围为空，则返回 p。若 p 为尾后迭代器，则函数行为未定义
emplace_after(p,*args*)	使用 *args* 在 p 指定的位置之后创建一个元素。返回一个指向这个新元素的迭代器。若 p 为尾后迭代器，则函数行为未定义
lst.erase_after(p) lst.erase_after(b,e)	删除 p 指向的位置之后的元素，或删除从 b 之后直到（但不包含）e 之间的元素。返回一个指向被删元素之后元素的迭代器，若不存在这样的元素，则返回尾后迭代器。如果 p 指向 lst 的尾元素或者是一个尾后迭代器，则函数行为未定义

当在 forward_list 中添加或删除元素时，我们必须关注两个迭代器——一个指向我们要处理的元素，另一个指向其前驱。例如，可以改写第 312 页中从 list 中删除奇数元素的循环程序，将其改为从 forward_list 中删除元素：

```
forward_list<int> flst = {0,1,2,3,4,5,6,7,8,9};
auto prev = flst.before_begin();          // 表示 flst 的“首前元素”
auto curr = flst.begin();                 // 表示 flst 中的第一个元素
while (curr != flst.end()) {              // 仍有元素要处理
    if (*curr % 2)                        // 若元素为奇数
        curr = flst.erase_after(prev);    // 删除它并移动 curr
    else {
        prev = curr;      // 移动迭代器 curr，指向下一个元素，prev 指向
```

```
            ++curr;        // curr之前的元素
        }
    }
```

此例中，curr 表示我们要处理的元素，prev 表示 curr 的前驱。调用 begin 来初始化 curr，这样第一步循环就会检查第一个元素是否是奇数。我们用 before_begin 来初始化 prev，它返回指向 curr 之前不存在的元素的迭代器。

当找到奇数元素后，我们将 prev 传递给 erase_after。此调用将 prev 之后的元素删除，即，删除 curr 指向的元素。然后我们将 curr 重置为 erase_after 的返回值，使得 curr 指向序列中下一个元素，prev 保持不变，仍指向（新）curr 之前的元素。如果 curr 指向的元素不是奇数，在 else 中我们将两个迭代器都向前移动。

352

9.3.4 节练习

练习 9.27：编写程序，查找并删除 forward_list<int>中的奇数元素。

练习 9.28：编写函数，接受一个 forward_list<string>和两个 string 共三个参数。函数应在链表中查找第一个 string，并将第二个 string 插入到紧接着第一个 string 之后的位置。若第一个 string 未在链表中，则将第二个 string 插入到链表末尾。

9.3.5 改变容器大小

如表 9.9 所描述，我们可以用 resize 来增大或缩小容器，与往常一样，array 不支持 resize。如果当前大小大于所要求的大小，容器后部的元素会被删除；如果当前大小小于新大小，会将新元素添加到容器后部：

```
list<int> ilist(10, 42);      // 10个int：每个的值都是42
ilist.resize(15);             // 将5个值为0的元素添加到ilist的末尾
ilist.resize(25, -1);         // 将10个值为-1的元素添加到ilist的末尾
ilist.resize(5);              // 从ilist末尾删除20个元素
```

resize 操作接受一个可选的元素值参数，用来初始化添加到容器中的元素。如果调用者未提供此参数，新元素进行值初始化（参见 3.3.1 节，第 88 页）。如果容器保存的是类类型元素，且 resize 向容器添加新元素，则我们必须提供初始值，或者元素类型必须提供一个默认构造函数。

表 9.9：顺序容器大小操作

resize 不适用于 array	
c.resize(n)	调整 c 的大小为 n 个元素。若 n<c.size()，则多出的元素被丢弃。若必须添加新元素，对新元素进行值初始化
c. resize(n,t)	调整 c 的大小为 n 个元素。任何新添加的元素都初始化为值 t

如果 resize 缩小容器，则指向被删除元素的迭代器、引用和指针都会失效；对 vector、string 或 deque 进行 resize 可能导致迭代器、指针和引用失效。