为了验证电话号码，我们需要访问模式的组成部分。例如，我们希望验证区号部分的数字如果用了左括号，那么它是否也在区号后面用了右括号。即，我们不希望出现(908.555.1800这样的号码。

为了获得匹配的组成部分，我们需要在定义正则表达式时使用子表达式。每个子表达式用一对括号包围：

```
// 整个正则表达式包含七个子表达式：( ddd )分隔符 ddd 分隔符 dddd
// 子表达式 1、3、4 和 6 是可选的；2、5 和 7 保存号码
"(\\()?(\\d{3})(\\))?([-. ])?(\\d{3})([-. ]?)(\\d{4})";
```

由于我们的模式使用了括号，而且必须去除反斜线的特殊含义，因此这个模式很难读（也很难写!）。理解此模式的最简单的方法是逐个剥离（括号包围的）子表达式：

1. (\\()?表示区号部分可选的左括号
2. (\\d{3})表示区号
3. (\\))?表示区号部分可选的右括号
4. ([-.])?表示区号部分可选的分隔符
5. (\\d{3})表示号码的下三位数字
6. ([-.])?表示可选的分隔符
7. (\\d{4})表示号码的最后四位数字

下面的代码读取一个文件，并用此模式查找与完整的电话号码模式匹配的数据。它会调用一个名为valid的函数来检查号码格式是否合法：

740

```
string phone =
    "(\\()?(\\d{3})(\\))?([-. ])?(\\d{3})([-. ]?)(\\d{4})";
regex r(phone); // regex 对象，用于查找我们的模式
smatch m;
string s;
// 从输入文件中读取每条记录
while (getline(cin, s)) {
    // 对每个匹配的电话号码
    for (sregex_iterator it(s.begin(), s.end(), r), end_it;
              it != end_it; ++it)
        // 检查号码的格式是否合法
        if (valid(*it))
            cout << "valid: " << it->str() << endl;
        else
            cout << "not valid: " << it->str() << endl;
}
```

使用子匹配操作

我们将使用表17.11中描述的子匹配操作来编写valid函数。需要记住的重要一点是，我们的 pattern 有七个子表达式。与往常一样，每个 smatch 对象会包含八个ssub_match元素。位置[0]的元素表示整个匹配；元素[1]...[7]表示每个对应的子表达式。

当调用 valid 时，我们知道已经有一个完整的匹配，但不知道每个可选的子表达式是否是匹配的一部分。如果一个子表达式是完整匹配的一部分，则其对应的 ssub_match 对象的matched成员为true。

表 17.11：子匹配操作

注意：这些操作适用于 ssub_match、csub_match、wssub_match、wcsub_match。

matched	一个 public bool 数据成员，指出此 ssub_match 是否匹配了
first	public 数据成员，指向匹配序列首元素和尾后位置的迭代器。如果未匹配，则 first 和 second 是相等的
second	
length()	匹配的大小。如果 matched 为 false，则返回 0
str()	返回一个包含输入中匹配部分的 string。如果 matched 为 false，则返回空 string
s = ssub	将 ssub_match 对象 ssub 转化为 string 对象 s。等价于 s=ssub.str()。转换运算符不是 explicit 的（参见 14.9.1 节，第 515 页）

741 在一个合法的电话号码中，区号要么是完整括号包围的，要么完全没有括号。因此，valid 要做什么工作依赖于号码是否以一个括号开始：

```
bool valid(const smatch& m)
{
    // 如果区号前有一个左括号
    if(m[1].matched)
        // 则区号后必须有一个右括号，之后紧跟剩余号码或一个空格
        return m[3].matched
               && (m[4].matched == 0 || m[4].str() == " ");
    else
        // 否则，区号后不能有右括号
        // 另两个组成部分间的分隔符必须匹配
        return !m[3].matched
               && m[4].str() == m[6].str();
}
```

我们首先检查第一个子表达式（即，左括号）是否匹配了。这个子表达式在 m[1] 中。如果匹配了，则号码是以左括号开始的。在此情况下，如果区号后的子表达式也匹配了（意味着区号后有右括号）则整个号码是合法的。而且，如果号码正确使用了括号，则下一个字符必须是一个空格或下一部分的第一个数字。

如果 m[1] 未匹配，（即，没有左括号），则区号后的子表达式也不应该匹配。如果它为空，则整个号码是合法的。

17.3.3 节练习

练习 17.19：为什么可以不先检查 m[4] 是否匹配了就直接调用 m[4].str()？

练习 17.20：编写你自己版本的验证电话号码的程序。

练习 17.21：使用本节中定义的 valid 函数重写 8.3.2 节（第 289 页）中的电话号码程序。

练习 17.22：重写你的电话号码程序，使之允许在号码的三个部分之间放置任意多个空白符。

练习 17.23：编写查找邮政编码的正则表达式。一个美国邮政编码可以由五位或九位数字组成。前五位数字和后四位数字之间可以用一个短横线分隔。

17.3.4 使用 regex_replace

正则表达式不仅用在我们希望查找一个给定序列的时候，还用在当我们想将找到的序列替换为另一个序列的时候。例如，我们可能希望将美国的电话号码转换为“ddd.ddd.dddd”的形式，即，区号和后面三位数字用一个点分隔。

当我们希望在输入序列中查找并替换一个正则表达式时，可以调用 **regex_replace**。表 17.12 描述了 regex_replace，类似搜索函数，它接受一个输入字符序列和一个 regex 对象，不同的是，它还接受一个描述我们想要的输出形式的字符串。 742

表 17.12：正则表达式替换操作

`m.format(dest, fmt, mft)` `m.format(fmt, mft)`	使用格式字符串 *fmt* 生成格式化输出，匹配在 m 中，可选的 match_flag_type 标志在 mft 中。第一个版本写入迭代器 dest 指向的目的位置（参见 10.5.1 节，第 365 页）并接受 *fmt* 参数，可以是一个 string，也可以是表示字符数组中范围的一对指针。第二个版本返回一个 string，保存输出，并接受 *fmt* 参数，可以是一个 string，也可以是一个指向空字符结尾的字符数组的指针。mft 的默认值为 format_default
`regex_replace(dest,seq, r, fmt, mft)` `regex_replace(seq,r,fmt,mft)`	遍历 *seq*，用 regex_search 查找与 regex 对象 r 匹配的子串。使用格式字符串 *fmt* 和可选的 match_flag_type 标志来生成输出。第一个版本将输出写入到迭代器 dest 指定的位置，并接受一对迭代器 *seq* 表示范围。第二个版本返回一个 string，保存输出，且 *seq* 既可以是一个 string 也可以是一个指向空字符结尾的字符数组的指针。在所有情况下，*fmt* 既可以是一个 string 也可以是一个指向空字符结尾的字符数组的指针，且 mft 的默认值为 match_default

替换字符串由我们想要的字符组合与匹配的子串对应的子表达式而组成。在本例中，我们希望在替换字符串中使用第二个、第五个和第七个子表达式。而忽略第一个、第三个、第四个和第六个子表达式，因为这些子表达式用来形成号码的原格式而非新格式中的一部分。我们用一个符号$后跟子表达式的索引号来表示一个特定的子表达式：

```
string fmt = "$2.$5.$7"; // 将号码格式改为 ddd.ddd.dddd
```

可以像下面这样使用我们的正则表达式模式和替换字符串：

```
regex r(phone); // 用来寻找模式的 regex 对象
string number = "(908) 555-1800";
cout << regex_replace(number, r, fmt) << endl;
```

此程序的输出为：

```
908.555.1800
```

只替换输入序列的一部分

正则表达式更有意思的一个用处是替换一个大文件中的电话号码。例如，我们有一个保存人名及其电话号码的文件：

743
```
morgan (201) 555-2368 862-555-0123
drew (973)555.0130
lee (609) 555-0132 2015550175 800.555-0000
```

我们希望将数据转换为下面这样：

```
morgan 201.555.2368 862.555.0123
drew 973.555.0130
lee 609.555.0132 201.555.0175 800.555.0000
```

可以用下面的程序完成这种转换：

```
int main()
{
    string phone =
       "(\\()?(\\d{3})(\\))?([-. ])?(\\d{3})([-. ])?(\\d{4})";
    regex r(phone); // 寻找模式所用的 regex 对象
    smatch m;
    string s;
    string fmt = "$2.$5.$7"; // 将号码格式改为 ddd.ddd.dddd
    // 从输入文件中读取每条记录
    while (getline(cin, s))
        cout << regex_replace(s, r, fmt) << endl;
    return 0;
}
```

我们读取每条记录，保存到 s 中，并将其传递给 regex_replace。此函数在输入序列中查找并转换所有匹配子串。

用来控制匹配和格式的标志

就像标准库定义标志来指导如何处理正则表达式一样，标准库还定义了用来在替换过程中控制匹配或格式的标志。表 17.13 列出了这些值。这些标志可以传递给函数 regex_search 或 regex_match 或是类 smatch 的 format 成员。

匹配和格式化标志的类型为 match_flag_type。这些值都定义在名为 regex_constants 的命名空间中。类似用于 bind 的 placeholders（参见 10.3.4 节，第 355 页），regex_constants 也是定义在命名空间 std 中的命名空间。为了使用 regex_constants 中的名字，我们必须在名字前同时加上两个命名空间的限定符：

```
using std::regex_constants::format_no_copy;
```

此声明指出，如果代码中使用了 format_no_copy，则表示我们想要使用命名空间 std::constants 中的这个名字。如下所示，我们也可以用另一种形式的 using 来代替上面的代码，我们将在 18.2.2 节（第 702 页）中介绍这种形式：

```
using namespace std::regex_constants;
```

744

表 17.13：匹配标志

定义在 regex_constants::match_flag_type 中	
match_default	等价于 format_default
match_not_bol	不将首字符作为行首处理
match_not_eol	不将尾字符作为行尾处理
match_not_bow	不将首字符作为单词首处理
match_not_eow	不将尾字符作为单词尾处理
match_any	如果存在多于一个匹配，则可返回任意一个匹配
match_not_null	不匹配任何空序列
match_continuous	匹配必须从输入的首字符开始
match_prev_avail	输入序列包含第一个匹配之前的内容
format_default	用 ECMAScript 规则替换字符串
format_sed	用 POSIX sed 规则替换字符串
format_no_copy	不输出输入序列中未匹配的部分
format_first_only	只替换子表达式的第一次出现

使用格式标志

默认情况下，regex_replace 输出整个输入序列。未与正则表达式匹配的部分会原样输出；匹配的部分按格式字符串指定的格式输出。我们可以通过在 regex_replace 调用中指定 format_no_copy 来改变这种默认行为：

```
// 只生成电话号码：使用新的格式字符串
string fmt2 = "$2.$5.$7 "; // 在最后一部分号码后放置空格作为分隔符
// 通知 regex_replace 只拷贝它替换的文本
cout << regex_replace(s, r, fmt2, format_no_copy) << endl;
```

给定相同的输入，此版本的程序生成

```
201.555.2368 862.555.0123
973.555.0130
609.555.0132 201.555.0175 800.555.0000
```

17.3.4 节练习

练习 17.24：编写你自己版本的重排电话号码格式的程序。

练习 17.25：重写你的电话号码程序，使之只输出每个人的第一个电话号码。

练习 17.26：重写你的电话号码程序，使之对多于一个电话号码的人只输出第二个和后续电话号码。

练习 17.27：编写程序，将九位数字邮政编码的格式转换为 ddddd-dddd。

17.4 随机数

745

C++11

程序通常需要一个随机数源。在新标准出现之前，C 和 C++都依赖于一个简单的 C 库函数 rand 来生成随机数。此函数生成均匀分布的伪随机整数，每个随机数的范围在 0 和一个系统相关的最大值（至少为 32767）之间。

rand 函数有一些问题：即使不是大多数，也有很多程序需要不同范围的随机数。一些应用需要随机浮点数。一些程序需要非均匀分布的数。而程序员为了解决这些问题而试图转换 rand 生成的随机数的范围、类型或分布时，常常会引入非随机性。

定义在头文件 random 中的随机数库通过一组协作的类来解决这些问题：**随机数引擎类**（random-number engines）和**随机数分布类**（random-number distribution）。表 17.14 描述了这些类。一个引擎类可以生成 unsigned 随机数序列，一个分布类使用一个引擎类生成指定类型的、在给定范围内的、服从特定概率分布的随机数。

表 17.14：随机数库的组成	
引擎	类型，生成随机 unsigned 整数序列
分布	类型，使用引擎返回服从特定概率分布的随机数

C++程序不应该使用库函数 rand，而应使用 default_random_engine 类和恰当的分布类对象。

17.4.1 随机数引擎和分布

随机数引擎是函数对象类（参见 14.8 节，第 506 页），它们定义了一个调用运算符，该运算符不接受参数并返回一个随机 unsigned 整数。我们可以通过调用一个随机数引擎对象来生成原始随机数：

```
default_random_engine e; // 生成随机无符号数
for (size_t i = 0; i < 10; ++i)
    // e() "调用"对象来生成下一个随机数
    cout << e() << " ";
```

在我们的系统中，此程序生成：

```
16807 282475249 1622650073 984943658 1144108930 470211272 ...
```

在本例中，我们定义了一个名为 e 的 **default_random_engine** 对象。在 for 循环内，我们调用对象 e 来获得下一个随机数。

746

标准库定义了多个随机数引擎类，区别在于性能和随机性质量不同。每个编译器都会指定其中一个作为 default_random_engine 类型。此类型一般具有最常用的特性。表 17.15 列出了随机数引擎操作，标准库定义的引擎类型列在附录 A.3.2（第 783 页）中。

表 17.15：随机数引擎操作	
Engine e;	默认构造函数；使用该引擎类型默认的种子
Engine e(s);	使用整型值 s 作为种子
e.seed(s)	使用种子 s 重置引擎的状态
e.min() e.max()	此引擎可生成的最小值和最大值
Engine::result_type	此引擎生成的 unsigned 整型类型
e.discard(u)	将引擎推进 u 步；u 的类型为 unsigned long long

对于大多数场合，随机数引擎的输出是不能直接使用的，这也是为什么早先我们称之为原始随机数。问题出在生成的随机数的值范围通常与我们需要的不符，而正确转换随机数的范围是极其困难的。

分布类型和引擎

为了得到在一个指定范围内的数，我们使用一个分布类型的对象：

```
// 生成 0 到 9 之间（包含）均匀分布的随机数
uniform_int_distribution<unsigned> u(0,9);
default_random_engine e; // 生成无符号随机整数
for (size_t i = 0; i < 10; ++i)
    // 将 u 作为随机数源
    // 每个调用返回在指定范围内并服从均匀分布的值
    cout << u(e) << " ";
```

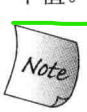
此代码生成下面这样的输出

```
0 1 7 4 5 2 0 6 6 9
```

此处我们将 u 定义为 uniform_int_distribution<unsigned>。此类型生成均匀分布的 unsigned 值。当我们定义一个这种类型的对象时，可以提供想要的最小值和最大值。在此程序中，u(0,9) 表示我们希望得到 0 到 9 之间（包含）的数。随机数分布类会使用包含的范围，从而我们可以得到给定整型类型的每个可能值。

类似引擎类型，分布类型也是函数对象类。分布类型定义了一个调用运算符，它接受一个随机数引擎作为参数。分布对象使用它的引擎参数生成随机数，并将其映射到指定的分布。

注意，我们传递给分布对象的是引擎对象本身，即 u(e)。如果我们将调用写成 u(e())，含义就变为将 e 生成的下一个值传递给 u，会导致一个编译错误。我们传递的是引擎本身，而不是它生成的下一个值，原因是某些分布可能需要调用引擎多次才能得到一个值。



当我们说随机数发生器时，是指分布对象和引擎对象的组合。

比较随机数引擎和 rand 函数

对熟悉 C 库函数 rand 的读者，值得注意的是：调用一个 default_random_engine 对象的输出类似 rand 的输出。随机数引擎生成的 unsigned 整数在一个系统定义的范围内，而 rand 生成的数的范围在 0 到 RAND_MAX 之间。一个引擎类型的范围可以通过调用该类型对象的 min 和 max 成员来获得：

747

```
cout << "min: " << e.min() << " max: " << e.max() << endl;
```

在我们的系统中，此程序生成下面的输出：

```
min: 1 max: 2147483646
```

引擎生成一个数值序列

随机数发生器有一个特性经常会使新手迷惑：即使生成的数看起来是随机的，但对一个给定的发生器，每次运行程序它都会返回相同的数值序列。序列不变这一事实在调试时非常有用。但另一方面，使用随机数发生器的程序也必须考虑这一特性。

作为一个例子，假定我们需要一个函数生成一个 vector，包含 100 个均匀分布在 0 到 9 之间的随机数。我们可能认为应该这样编写此函数：

```
// 几乎肯定是生成随机整数 vector 的错误方法
// 每次调用这个函数都会生成相同的 100 个数!
vector<unsigned> bad_randVec()
{
    default_random_engine e;
    uniform_int_distribution<unsigned> u(0,9);
    vector<unsigned> ret;
    for (size_t i = 0; i < 100; ++i)
        ret.push_back(u(e));
    return ret;
}
```

但是，每次调用这个函数都会返回相同的 vector：

```
vector<unsigned> v1(bad_randVec());
vector<unsigned> v2(bad_randVec());
// 将打印"equal"
cout << ((v1 == v2) ? "equal" : "not equal") << endl;
```

748 此代码会打印 equal，因为 vector v1 和 v2 具有相同的值。

编写此函数的正确方法是将引擎和关联的分布对象定义为 static 的（参见 6.1.1 节，第 185 页）：

```
// 返回一个 vector，包含 100 个均匀分布的随机数
vector<unsigned> good_randVec()
{
    // 由于我们希望引擎和分布对象保持状态，因此应该将它们
    // 定义为 static 的，从而每次调用都生成新的数
    static default_random_engine e;
    static uniform_int_distribution<unsigned> u(0,9);
    vector<unsigned> ret;
    for (size_t i = 0; i < 100; ++i)
        ret.push_back(u(e));
    return ret;
}
```

由于 e 和 u 是 static 的，因此它们在函数调用之间会保持住状态。第一次调用会使用 u(e) 生成的序列中的前 100 个随机数，第二次调用会获得接下来 100 个，依此类推。

一个给定的随机数发生器一直会生成相同的随机数序列。一个函数如果定义了局部的随机数发生器，应该将其（包括引擎和分布对象）定义为 static 的。否则，每次调用函数都会生成相同的序列。

设置随机数发生器种子

随机数发生器会生成相同的随机数序列这一特性在调试中很有用。但是，一旦我们的程序调试完毕，我们通常希望每次运行程序都会生成不同的随机结果，可以通过提供一个**种子**（seed）来达到这一目的。种子就是一个数值，引擎可以利用它从序列中一个新位置重新开始生成随机数。

为引擎设置种子有两种方式：在创建引擎对象时提供种子，或者调用引擎的 seed 成员：

```
default_random_engine e1;                    // 使用默认种子
default_random_engine e2(2147483646);        // 使用给定的种子值
// e3 和 e4 将生成相同的序列，因为它们使用了相同的种子
default_random_engine e3;                    // 使用默认种子值
e3.seed(32767);                              // 调用 seed 设置一个新种子值
default_random_engine e4(32767);             // 将种子值设置为 32767
for (size_t i = 0; i != 100; ++i) {
    if (e1() == e2())
        cout << "unseeded match at iteration: " << i << endl;
    if (e3() != e4())
        cout << "seeded differs at iteration: " << i << endl;
}
```

本例中我们定义了四个引擎。前两个引擎 e1 和 e2 的种子不同，因此应该生成不同的序列。后两个引擎 e3 和 e4 有相同的种子，它们将生成相同的序列。 749

选择一个好的种子，与生成好的随机数所涉及的其他大多数事情相同，是极其困难的。可能最常用的方法是调用系统函数 time。这个函数定义在头文件 ctime 中，它返回从一个特定时刻到当前经过了多少秒。函数 time 接受单个指针参数，它指向用于写入时间的数据结构。如果此指针为空，则函数简单地返回时间：

```
default_random_engine e1(time(0)); // 稍微随机些的种子
```

由于 time 返回以秒计的时间，因此这种方式只适用于生成种子的间隔为秒级或更长的应用。

如果程序作为一个自动过程的一部分反复运行，将 time 的返回值作为种子的方式就无效了；它可能多次使用的都是相同的种子。

17.4.1 节练习

练习 17.28：编写函数，每次调用生成并返回一个均匀分布的随机 unsigned int。

练习 17.29：修改上一题中编写的函数，允许用户提供一个种子作为可选参数。

练习 17.30：再次修改你的程序，此次再增加两个参数，表示函数允许返回的最小值和最大值。

17.4.2 其他随机数分布

随机数引擎生成 unsigned 数，范围内的每个数被生成的概率都是相同的。而应用程序常常需要不同类型或不同分布的随机数。标准库通过定义不同随机数分布对象来满足这两方面的要求，分布对象和引擎对象协同工作，生成要求的结果。表 17.16 列出了分布类型所支持的操作。

生成随机实数

程序常需要一个随机浮点数的源。特别是，程序经常需要 0 到 1 之间的随机数。

最常用但不正确的从 rand 获得一个随机浮点数的方法是用 rand() 的结果除以 RAND_MAX，即，系统定义的 rand 可以生成的最大随机数的上界。这种方法不正确的原因是随机整数的精度通常低于随机浮点数，这样，有一些浮点值就永远不会被生成了。

使用新标准库设施，可以很容易地获得随机浮点数。我们可以定义一个 750

uniform_real_distribution 类型的对象，并让标准库来处理从随机整数到随机浮点数的映射。与处理 uniform_int_distribution 一样，在定义对象时，我们指定最小值和最大值：

```
default_random_engine e; // 生成无符号随机整数
// 0 到 1（包含）的均匀分布
uniform_real_distribution<double> u(0,1);
for (size_t i = 0; i < 10; ++i)
     cout << u(e) << " ";
```

这段代码与之前生成 unsigned 值的程序几乎相同。但是，由于我们使用了一个不同的分布类型，此版本会生成不同的结果：

```
0.131538 0.45865 0.218959 0.678865 0.934693 0.519416 ...
```

表 17.16：分布类型的操作

Dist d;	默认构造函数；使 d 准备好被使用。 其他构造函数依赖于 *Dist* 的类型；参见附录 A.3 节（第 781 页）。 分布类型的构造函数是 explicit 的（参见 7.5.4 节，第 265 页）
d(e)	用相同的 e 连续调用 d 的话，会根据 d 的分布式类型生成一个随机数序列；e 是一个随机数引擎对象
d.min() d.max()	返回 d(e) 能生成的最小值和最大值
d.reset()	重建 d 的状态，使得随后对 d 的使用不依赖于 d 已经生成的值

使用分布的默认结果类型

分布类型都是模板，具有单一的模板类型参数，表示分布生成的随机数的类型，对此有一个例外，我们将在 17.4.2 节（第 665 页）中进行介绍。这些分布类型要么生成浮点类型，要么生成整数类型。

每个分布模板都有一个默认模板实参（参见 16.1.3 节，第 594 页）。生成浮点值的分布类型默认生成 double 值，而生成整型值的分布默认生成 int 值。由于分布类型只有一个模板参数，因此当我们希望使用默认随机数类型时要记得在模板名之后使用空尖括号（参见 16.1.3 节，第 594 页）：

```
// 空<>表示我们希望使用默认结果类型
uniform_real_distribution<> u(0,1); // 默认生成 double 值
```

751

生成非均匀分布的随机数

除了正确生成在指定范围内的数之外，新标准库的另一个优势是可以生成非均匀分布的随机数。实际上，新标准库定义了 20 种分布类型，这些类型列在附录 A.3（第 781）中。

作为一个例子，我们将生成一个正态分布的值的序列，并画出值的分布。由于 normal_distribution 生成浮点值，我们的程序使用头文件 cmath 中的 lround 函数将每个随机数舍入到最接近的整数。我们将生成 200 个数，它们以均值 4 为中心，标准差为 1.5。由于使用的是正态分布，我们期望生成的数中大约 99%都在 0 到 8 之间（包含）。我们的程序会对这个范围内的每个整数统计有多少个生成的数映射到它：

```
default_random_engine e;            // 生成随机整数
normal_distribution<> n(4,1.5);     // 均值 4，标准差 1.5
```

```
vector<unsigned> vals(9);          // 9个元素均为0
for (size_t i = 0; i != 200; ++i) {
    unsigned v = lround(n(e));     // 舍入到最接近的整数
    if (v < vals.size())           // 如果结果在范围内
        ++vals[v];                 // 统计每个数出现了多少次
}
for (size_t j = 0; j != vals.size(); ++j)
    cout << j << ": " << string(vals[j], '*') << endl;
```

我们首先定义了随机数发生器对象和一个名为 vals 的 vector。我们用 vals 来统计范围 0…8 中的每个数出现了多少次。与我们使用 vector 的大多数程序不同，此程序按需求大小为 vals 分配空间，每个元素都被初始化为 0。

在 for 循环中，我们调用 lround(n(e)) 来将 n(e) 返回的值舍入到最接近的整数。获得浮点随机数对应的整数后，我们将它作为计数器 vector 的下标。由于 n(e) 可能生成范围 0 到 8 之外的数，所以我们首先检查生成的数是否在范围内，然后再将其作为 vals 的下标。如果结果确实在范围内，我们递增对应的计数器。

当循环结束时，我们打印 vals 的内容，可能会打印出像下面这样的结果：

```
0: ***
1: ********
2: ********************
3: ****************************************
4: ************************************************************
5: ******************************************
6: ************************
7: *******
8: *
```

本例中我们打印一个由星号组成的 string，有多少随机数等于此下标我们就打印多少个星号。注意，此图并不是完美对称的。如果打印出的图是完美对称的，我们反倒有理由怀疑随机数发生器的质量了。 752

bernoulli_distribution 类

我们注意到有一个分布不接受模板参数，即 bernoulli_distribution，因为它是一个普通类，而非模板。此分布总是返回一个 bool 值。它返回 true 的概率是一个常数，此概率的默认值是 0.5。

作为一个这种分布的例子，我们可以编写一个程序，这个程序与用户玩一个游戏。为了进行这个游戏，其中一个游戏者——用户或是程序——必须先行。我们可以用一个值范围是 0 到 1 的 uniform_int_distribution 来选择先行的游戏者，但也可以用伯努利分布来完成这个选择。假定已有一个名为 play 的函数来进行游戏，我们可以编写像下面这样的循环来与用户交互：

```
string resp;
default_random_engine e; // e应保持状态，所以必须在循环外定义！
bernoulli_distribution b;// 默认是50/50的机会
do {
    bool first = b(e);    // 如果为true，则程序先行
    cout << (first ? "We go first"
                   : "You get to go first") << endl;
```

```
        // 传递谁先行的指示，进行游戏
        cout << ((play(first)) ? "sorry, you lost"
                               : "congrats, you won") << endl;
        cout << "play again? Enter 'yes' or 'no'" << endl;
    } while (cin >> resp && resp[0] == 'y');
```

我们用一个 do while 循环（参见 5.4.4 节，第 169 页）来反复提示用户进行游戏。

由于引擎返回相同的随机数序列（参见 17.4.1 节，第 661 页），所以我们必须在循环外声明引擎对象。否则，每步循环都会创建一个新引擎，从而每步循环都会生成相同的值。类似的，分布对象也要保持状态，因此也应该在循环外定义。

在此程序中使用 bernoulli_distribution 的一个原因是它允许我们调整选择先行一方的概率：

```
bernoulli_distribution b(.55); // 给程序一个微小的优势
```

如果 b 定义如上，则程序有 55/45 的机会先行。

17.4.2 节练习

练习 17.31：对于本节中的游戏程序，如果在 do 循环内定义 b 和 e，会发生什么？

练习 17.32：如果我们在循环内定义 resp，会发生什么？

练习 17.33：修改 11.3.6 节（第 392 页）中的单词转换程序，允许对一个给定单词有多种转换方式，每次随机选择一种进行实际转换。

17.5 IO 库再探

在第 8 章中我们介绍了 IO 库的基本结构及其最常用的部分。在本节中，我们将介绍三个更特殊的 IO 库特性：格式控制、未格式化 IO 和随机访问。

753

17.5.1 格式化输入与输出

除了条件状态外（参见 8.1.2 节，第 279 页），每个 iostream 对象还维护一个格式状态来控制 IO 如何格式化的细节。格式状态控制格式化的某些方面，如整型值是几进制、浮点值的精度、一个输出元素的宽度等。

标准库定义了一组**操纵符**（manipulator）（参见 1.2 节，第 6 页）来修改流的格式状态，如表 17.7 和表 17.8 所示。一个操纵符是一个函数或是一个对象，会影响流的状态，并能用作输入或输出运算符的运算对象。类似输入和输出运算符，操纵符也返回它所处理的流对象，因此我们可以在一条语句中组合操纵符和数据。

我们已经在程序中使用过一个操纵符——endl，我们将它“写”到输出流，就像它是一个值一样。但 endl 不是一个普通值，而是一个操作：它输出一个换行符并刷新缓冲区。

很多操纵符改变格式状态

操纵符用于两大类输出控制：控制数值的输出形式以及控制补白的数量和位置。大多数改变格式状态的操纵符都是设置/复原成对的：一个操纵符用来将格式状态设置为一个新值，而另一个用来将其复原，恢复为正常的默认格式。

当操纵符改变流的格式状态时，通常改变后的状态对所有后续 IO 都生效。

当我们有一组 IO 操作希望使用相同的格式时，操纵符对格式状态的改变是持久的这一特性很有用。实际上，一些程序会利用操纵符的这一特性对其所有输入或输出重置一个或多个格式规则的行为。在这种情况下，操纵符会改变流这一特性就是满足要求的了。

但是，很多程序（而且更重要的是，很多程序员）期望流的状态符合标准库正常的默认设置。在这些情况下，将流的状态置于一个非标准状态可能会导致错误。因此，通常最好在不再需要特殊格式时尽快将流恢复到默认状态。

控制布尔值的格式

754

操纵符改变对象的格式状态的一个例子是 boolalpha 操纵符。默认情况下，bool 值打印为 1 或 0。一个 true 值输出为整数 1，而 false 输出为 0。我们可以通过对流使用 boolalpha 操纵符来覆盖这种格式：

```
cout << "default bool values: " << true << " " << false
     << "\nalpha bool values: " << boolalpha
     << true << " " << false << endl;
```

执行这段程序会得到下面的结果：

```
default bool values: 1 0
alpha bool values: true false
```

一旦向 cout “写入”了 boolalpha，我们就改变了 cout 打印 bool 值的方式。后续打印 bool 值的操作都会打印 true 或 false 而非 1 或 0。

为了取消 cout 格式状态的改变，我们使用 noboolalpha：

```
bool bool_val = get_status();
cout << boolalpha     // 设置 cout 的内部状态
<< bool_val
<< noboolalpha;       // 将内部状态恢复为默认格式
```

本例中我们改变了 bool 值的格式，但只对 bool_val 的输出有效。一旦完成此值的打印，我们立即将流恢复到初始状态。

指定整型值的进制

默认情况下，整型值的输入输出使用十进制。我们可以使用操纵符 hex、oct 和 dec 将其改为十六进制、八进制或是改回十进制：

```
cout << "default: " << 20 << " " << 1024 << endl;
cout << "octal: " << oct << 20 << " " << 1024 << endl;
cout << "hex: " << hex << 20 << " " << 1024 << endl;
cout << "decimal: " << dec << 20 << " " << 1024 << endl;
```

当编译并执行这段程序时，会得到如下输出：

```
default: 20 1024
octal: 24 2000
hex: 14 400
decimal: 20 1024
```

注意，类似 boolalpha，这些操纵符也会改变格式状态。它们会影响下一个和随后所有的整型输出，直至另一个操纵符又改变了格式为止。

操纵符 hex、oct 和 dec 只影响整型运算对象，浮点值的表示形式不受影响。

755 在输出中指出进制

默认情况下，当我们打印出数值时，没有可见的线索指出使用的是几进制。例如，20 是十进制的 20 还是 16 的八进制表示？当我们按十进制打印数值时，打印结果会符合我们的期望。如果需要打印八进制值或十六进制值，应该使用 showbase 操纵符。当对流应用 showbase 操纵符时，会在输出结果中显示进制，它遵循与整型常量中指定进制相同的规范：

- 前导 0x 表示十六进制。
- 前导 0 表示八进制。
- 无前导字符串表示十进制。

我们可以使用 showbase 修改前一个程序：

```
cout << showbase; // 当打印整型值时显示进制
cout << "default: " << 20 << " " << 1024 << endl;
cout << "in octal: " << oct << 20 << " " << 1024 << endl;
cout << "in hex: " << hex << 20 << " " << 1024 << endl;
cout << "in decimal: " << dec << 20 << " " << 1024 << endl;
cout << noshowbase; // 恢复流状态
```

修改后的程序的输出会更清楚地表明底层值到底是什么：

```
default: 20 1024
in octal: 024 02000
in hex: 0x14 0x400
in decimal: 20 1024
```

操纵符 noshowbase 恢复 cout 的状态，从而不再显示整型值的进制。

默认情况下，十六进制值会以小写打印，前导字符也是小写的 x。我们可以通过使用 uppercase 操纵符来输出大写的 X 并将十六进制数字 a-f 以大写输出：

```
cout << uppercase << showbase << hex
     << "printed in hexadecimal: " << 20 << " " << 1024
     << nouppercase << noshowbase << dec << endl;
```

这条语句生成如下输出：

```
printed in hexadecimal: 0X14 0X400
```

我们使用了操纵符 nouppercase、noshowbase 和 dec 来重置流的状态。

控制浮点数格式

我们可以控制浮点数输出三个种格式：

- 以多高精度（多少个数字）打印浮点值
- 数值是打印为十六进制、定点十进制还是科学记数法形式
- 对于没有小数部分的浮点值是否打印小数点

756

默认情况下，浮点值按六位数字精度打印；如果浮点值没有小数部分，则不打印小数点；根据浮点数的值选择打印成定点十进制或科学记数法形式。标准库会选择一种可读性更好的格式：非常大和非常小的值打印为科学记数法形式，其他值打印为定点十进制形式。

指定打印精度

默认情况下，精度会控制打印的数字的总数。当打印时，浮点值按当前精度舍入而非截断。因此，如果当前精度为四位数字，则 3.14159 将打印为 3.142；如果精度为三位数字，则打印为 3.14。

我们可以通过调用 IO 对象的 precision 成员或使用 setprecision 操纵符来改变精度。precision 成员是重载的（参见 6.4 节，第 206 页）。一个版本接受一个 int 值，将精度设置为此值，并返回旧精度值。另一个版本不接受参数，返回当前精度值。setprecision 操纵符接受一个参数，用来设置精度。

操纵符 setprecision 和其他接受参数的操纵符都定义在头文件 iomanip 中。

下面的程序展示了控制浮点值打印精度的不同方法：

```
// cout.precision 返回当前精度值
cout << "Precision: " << cout.precision()
     << ", Value: "   << sqrt(2.0) << endl;
// cout.precision(12)将打印精度设置为 12 位数字
cout.precision(12);
cout << "Precision: " << cout.precision()
     << ", Value: "   << sqrt(2.0) << endl;
// 另一种设置精度的方法是使用 setprecision 操纵符
cout << setprecision(3);
cout << "Precision: " << cout.precision()
     << ", Value: "   << sqrt(2.0) << endl;
```

编译并执行这段程序，会得到如下输出：

```
Precision: 6, Value: 1.41421
Precision: 12, Value: 1.41421356237
Precision: 3, Value: 1.41
```

此程序调用标准库 sqrt 函数，它定义在头文件 cmath 中。sqrt 函数是重载的，不同版本分别接受一个 float、double 或 long double 参数，返回实参的平方根。

757

表 17.17：定义在 iostream 中的操纵符

boolalpha	将 true 和 false 输出为字符串
* noboolalpha	将 true 和 false 输出为 1，0
showbase	对整型值输出表示进制的前缀
* noshowbase	不生成表示进制的前缀
showpoint	对浮点值总是显示小数点

续表

*	noshowpoint	只有当浮点值包含小数部分时才显示小数点
	showpos	对非负数显示+
*	noshowpos	对非负数不显示+
	uppercase	在十六进制值中打印 0X，在科学记数法中打印 E
*	nouppercase	在十六进制值中打印 0x，在科学记数法中打印 e
*	dec	整型值显示为十进制
	hex	整型值显示为十六进制
	oct	整型值显示为八进制
	left	在值的右侧添加填充字符
	right	在值的左侧添加填充字符
	internal	在符号和值之间添加填充字符
	fixed	浮点值显示为定点十进制
	scientific	浮点值显示为科学记数法
	hexfloat	浮点值显示为十六进制（C++11 新特性）
	defaultfloat	重置浮点数格式为十进制（C++11 新特性）
	unitbuf	每次输出操作后都刷新缓冲区
*	nounitbuf	恢复正常的缓冲区刷新方式
*	skipws	输入运算符跳过空白符
	noskipws	输入运算符不跳过空白符
	flush	刷新 ostream 缓冲区
	ends	插入空字符，然后刷新 ostream 缓冲区
	endl	插入换行，然后刷新 ostream 缓冲区

* 表示默认流状态

指定浮点数记数法

Best Practices 除非你需要控制浮点数的表示形式（如，按列打印数据或打印表示金额或百分比的数据），否则由标准库选择记数法是最好的方式。

通过使用恰当的操纵符，我们可以强制一个流使用科学记数法、定点十进制或是十六进制记数法。操纵符 scientific 改变流的状态来使用科学记数法。操纵符 fixed 改变流的状态来使用定点十进制。

C++ 11 在新标准库中，通过使用 hexfloat 也可以强制浮点数使用十六进制格式。新标准库还提供另一个名为 defaultfloat 的操纵符，它将流恢复到默认状态——根据要打印的值选择记数法。

758

这些操纵符也会改变流的精度的默认含义。在执行 scientific、fixed 或 hexfloat 后，精度值控制的是小数点后面的数字位数，而默认情况下精度值指定的是数字的总位数——既包括小数点之后的数字也包括小数点之前的数字。使用 fixed 或 scientific 令我们可以按列打印数值，因为小数点距小数部分的距离是固定的：

```
cout << "default format: " << 100 * sqrt(2.0) << '\n'
     << "scientific: " << scientific << 100 * sqrt(2.0) << '\n'
     << "fixed decimal: " << fixed << 100 * sqrt(2.0) << '\n'
     << "hexadecimal: " << hexfloat << 100 * sqrt(2.0) << '\n'
     << "use defaults: " << defaultfloat << 100 * sqrt(2.0)
```

```
        << "\n\n";
```

此程序会生成下面的输出：

```
default format: 141.421
scientific: 1.414214e+002
fixed decimal: 141.421356
hexadecimal: 0x1.1ad7bcp+7
use defaults: 141.421
```

默认情况下，十六进制数字和科学记数法中的 e 都打印成小写形式。我们可以用 uppercase 操纵符打印这些字母的大写形式。

打印小数点

默认情况下，当一个浮点值的小数部分为 0 时，不显示小数点。showpoint 操纵符强制打印小数点：

```
cout << 10.0 << endl;          // 打印 10
cout << showpoint << 10.0      // 打印 10.0000
     << noshowpoint << endl;   // 恢复小数点的默认格式
```

操纵符 noshowpoint 恢复默认行为。下一个输出表达式将有默认行为，即，当浮点值的小数部分为 0 时不输出小数点。

输出补白

当按列打印数据时，我们常常需要非常精细地控制数据格式。标准库提供了一些操纵符帮助我们完成所需的控制：

- setw 指定下一个数字或字符串值的最小空间。
- left 表示左对齐输出。
- right 表示右对齐输出，右对齐是默认格式。
- internal 控制负数的符号的位置，它左对齐符号，右对齐值，用空格填满所有中间空间。 759
- setfill 允许指定一个字符代替默认的空格来补白输出。

setw 类似 endl，不改变输出流的内部状态。它只决定下一个输出的大小。

下面程序展示了如何使用这些操纵符：

```
int i = -16;
double d = 3.14159;
// 补白第一列，使用输出中最小 12 个位置
cout << "i: " << setw(12) << i << "next col" << '\n'
     << "d: " << setw(12) << d << "next col" << '\n';
// 补白第一列，左对齐所有列
cout << left
     << "i: " << setw(12) << i << "next col" << '\n'
     << "d: " << setw(12) << d << "next col" << '\n'
     << right; // 恢复正常对齐
// 补白第一列，右对齐所有列
cout << right
     << "i: " << setw(12) << i << "next col" << '\n'
```

```
        << "d: " << setw(12) << d << "next col" << '\n';
    // 补白第一列，但补在域的内部
    cout << internal
        << "i: " << setw(12) << i << "next col" << '\n'
        << "d: " << setw(12) << d << "next col" << '\n';
    // 补白第一列，用#作为补白字符
    cout << setfill('#')
        << "i: " << setw(12) << i << "next col" << '\n'
        << "d: " << setw(12) << d << "next col" << '\n'
        << setfill(' '); // 恢复正常的补白字符
```

执行这段程序，会得到下面的输出：

```
i:           -16next col
d:       3.14159next col
i: -16           next col
d: 3.14159       next col
i:           -16next col
d:       3.14159next col
i: -           16next col
d:       3.14159next col
i:  -#########16next col
d:  #####3.14159next col
```

760

表 17.18：定义在 iomanip 中的操纵符

setfill(ch)	用 ch 填充空白
setprecision(n)	将浮点精度设置为 n
setw(w)	读或写值的宽度为 w 个字符
setbase(b)	将整数输出为 b 进制

控制输入格式

默认情况下，输入运算符会忽略空白符（空格符、制表符、换行符、换纸符和回车符）。下面的循环

```
char ch;
while (cin >> ch)
    cout << ch;
```

当给定下面输入序列时

```
a b     c
d
```

循环会执行 4 次，读取字符 a 到 d，跳过中间的空格以及可能的制表符和换行符。此程序的输出是

```
abcd
```

操纵符 noskipws 会令输入运算符读取空白符，而不是跳过它们。为了恢复默认行为，我们可以使用 skipws 操纵符：

```
cin >> noskipws; // 设置 cin 读取空白符
while (cin >> ch)
    cout << ch;
```

```
cin >> skipws; // 将 cin 恢复到默认状态，从而丢弃空白符
```

给定与前一个程序相同的输入，此循环会执行 7 次，从输入中既读取普通字符又读取空白符。此循环的输出为

```
a b     c
d
```

17.5.1 节练习

练习 17.34：编写一个程序，展示如何使用表 17.17 和表 17.18 中的每个操纵符。

练习 17.35：修改第 670 页中的程序，打印 2 的平方根，但这次打印十六进制数字的大写形式。

练习 17.36：修改上一题中的程序，打印不同的浮点数，使它们排成一列。

17.5.2 未格式化的输入/输出操作

761

到目前为止，我们的程序只使用过**格式化 IO**（formatted IO）操作。输入和输出运算符（<<和>>）根据读取或写入的数据类型来格式化它们。输入运算符忽略空白符，输出运算符应用补白、精度等规则。

标准库还提供了一组低层操作，支持**未格式化 IO**（unformatted IO）。这些操作允许我们将一个流当作一个无解释的字节序列来处理。

单字节操作

有几个未格式化操作每次一个字节地处理流。这些操作列在表 17.19 中，它们会读取而不是忽略空白符。例如，我们可以使用未格式化 IO 操作 get 和 put 来读取和写入一个字符：

```
char ch;
while (cin.get(ch))
        cout.put(ch);
```

此程序保留输入中的空白符，其输出与输入完全相同。它的执行过程与前一个使用 noskipws 的程序完全相同。

表 17.19：单字节低层 IO 操作

is.get(ch)	从 istream is 读取下一个字节存入字符 ch 中。返回 is
os.put(ch)	将字符 ch 输出到 ostream os。返回 os
is.get()	将 is 的下一个字节作为 int 返回
is.putback(ch)	将字符 ch 放回 is。返回 is
is.unget()	将 is 向后移动一个字节。返回 is
is.peek()	将下一个字节作为 int 返回，但不从流中删除它

将字符放回输入流

有时我们需要读取一个字符才能知道还未准备好处理它。在这种情况下，我们希望将字符放回流中。标准库提供了三种方法退回字符，它们有着细微的差别：

- peek 返回输入流中下一个字符的副本，但不会将它从流中删除，peek 返回的值仍然留在流中。

- unget 使得输入流向后移动，从而最后读取的值又回到流中。即使我们不知道最后从流中读取什么值，仍然可以调用 unget。
- putback 是更特殊版本的 unget：它退回从流中读取的最后一个值，但它接受一个参数，此参数必须与最后读取的值相同。

762 一般情况下，在读取下一个值之前，标准库保证我们可以退回最多一个值。即，标准库不保证在中间不进行读取操作的情况下能连续调用 putback 或 unget。

从输入操作返回的 int 值

函数 peek 和无参的 get 版本都以 int 类型从输入流返回一个字符。这有些令人吃惊，可能这些函数返回一个 char 看起来会更自然。

这些函数返回一个 int 的原因是：可以返回文件尾标记。我们使用 char 范围中的每个值来表示一个真实字符，因此，取值范围中没有额外的值可以用来表示文件尾。

返回 int 的函数将它们要返回的字符先转换为 unsigned char，然后再将结果提升到 int。因此，即使字符集中有字符映射到负值，这些操作返回的 int 也是正值（参见 2.1.2 节，第 32 页）。而标准库使用负值表示文件尾，这样就可以保证与任何合法字符的值都不同。头文件 cstdio 定义了一个名为 EOF 的 const，我们可以用它来检测从 get 返回的值是否是文件尾，而不必记忆表示文件尾的实际数值。对我们来说重要的是，用一个 int 来保存从这些函数返回的值：

```
int ch; // 使用一个 int，而不是一个 char 来保存 get() 的返回值
// 循环读取并输出输入中的所有数据
while ((ch = cin.get()) != EOF)
        cout.put(ch);
```

此程序与第 673 页中的程序完成相同的工作，唯一的不同是用来读取输入的 get 版本不同。

多字节操作

一些未格式化 IO 操作一次处理大块数据。如果速度是要考虑的重点问题的话，这些操作是很重要的，但类似其他低层操作，这些操作也容易出错。特别是，这些操作要求我们自己分配并管理用来保存和提取数据的字符数组（参见 12.2 节，第 423 页）。表 17.20 列出了多字节操作。

表 17.20：多字节低层 IO 操作

is.get(sink, size, delim) 从 is 中读取最多 size 个字节，并保存在字符数组中，字符数组的起始地址由 sink 给出。读取过程直至遇到字符 delim 或读取了 size 个字节或遇到文件尾时停止。如果遇到了 delim，则将其留在输入流中，不读取出来存入 sink
is.getline(sink, size, delim) 与接受三个参数的 get 版本类似，但会读取并丢弃 delim
is.read(sink, size) 读取最多 size 个字节，存入字符数组 sink 中。返回 is
is.gcount() 返回上一个未格式化读取操作从 is 读取的字节数
os.write(source, size) 将字符数组 source 中的 size 个字节写入 os。返回 os

续表

is.ignore(size, delim) 读取并忽略最多 size 个字符，包括 delim。与其他未格式化函数不同，ignore 有默认参数：size 的默认值为 1，delim 的默认值为文件尾

get 和 getline 函数接受相同的参数，它们的行为类似但不相同。在两个函数中，sink 都是一个 char 数组，用来保存数据。两个函数都一直读取数据，直至下面条件之一发生：

- 已读取了 size-1 个字符
- 遇到了文件尾
- 遇到了分隔符

两个函数的差别是处理分隔符的方式：get 将分隔符留作 istream 中的下一个字符，而 getline 则读取并丢弃分隔符。无论哪个函数都不会将分隔符保存在 sink 中。

一个常见的错误是本想从流中删除分隔符，但却忘了做。

763

确定读取了多少个字符

某些操作从输入读取未知个数的字节。我们可以调用 gcount 来确定最后一个未格式化输入操作读取了多少个字符。应该在任何后续未格式化输入操作之前调用 gcount。特别是，将字符退回流的单字符操作也属于未格式化输入操作。如果在调用 gcount 之前调用了 peek、unget 或 putback，则 gcount 的返回值为 0。

小心：低层函数容易出错

一般情况下，我们主张使用标准库提供的高层抽象。返回 int 的 IO 操作很好地解释了原因。

一个常见的编程错误是将 get 或 peek 的返回值赋予一个 char 而不是一个 int。这样做是错误的，但编译器却不能发现这个错误。最终会发生什么依赖于程序运行于哪台机器以及输入数据是什么。例如，在一台 char 被实现为 unsigned char 的机器上，下面的循环永远不会停止：

```
char ch; // 此处使用 char 就是引入灾难!
// 从 cin.get 返回的值被转换为 char，然后与一个 int 比较
while ((ch = cin.get()) != EOF)
        cout.put(ch);
```

问题出在当 get 返回 EOF 时，此值会被转换为一个 unsigned char。转换得到的值与 EOF 的 int 值不再相等，因此循环永远也不会停止。这种错误很可能在调试时发现。

在一台 char 被实现为 signed char 的机器上，我们不能确定循环的行为。当一个越界的值被赋予一个 signed 变量时会发生什么完全取决于编译器。在很多机器上，这个循环可以正常工作，除非输入序列中有一个字符与 EOF 值匹配。虽然在普通数据中这种字符不太可能出现，但低层 IO 通常用于读取二进制值的场合，而这些二进制值不能直接映射到普通字符和数值。例如，在我们的机器上，如果输入中包含有一个值为'\377'的字符，则循环会提前终止。因为在我们的机器上，将-1 转换为一个 signed char，就会得到'\377'。如果输入中有这个值，则它会被（过早）当作文件尾指示符。

当我们读写有类型的值时，这种错误就不会发生。如果你可以使用标准库提供的类型更加安全、更高层的操作，就应该使用它们。

17.5.2 节练习

练习 17.37：用未格式化版本的 getline 逐行读取一个文件。测试你的程序，给它一个文件，既包含空行又包含长度超过你传递给 getline 的字符数组大小的行。

练习 17.38：扩展上一题中你的程序，将读入的每个单词打印到它所在的行。

17.5.3 流随机访问

各种流类型通常都支持对流中数据的随机访问。我们可以重定位流，使之跳过一些数据，首先读取最后一行，然后读取第一行，依此类推。标准库提供了一对函数，来定位（seek）到流中给定的位置，以及告诉（tell）我们当前位置。

随机 IO 本质上是依赖于系统的。为了理解如何使用这些特性，你必须查询系统文档。

虽然标准库为所有流类型都定义了 seek 和 tell 函数，但它们是否会做有意义的事情依赖于流绑定到哪个设备。在大多数系统中，绑定到 cin、cout、cerr 和 clog 的流
764 不支持随机访问——毕竟，当我们向 cout 直接输出数据时，类似向回跳十个位置这种操作是没有意义的。对这些流我们可以调用 seek 和 tell 函数，但在运行时会出错，将流置于一个无效状态。

由于 istream 和 ostream 类型通常不支持随机访问，所以本节剩余内容只适用于 fstream 和 sstream 类型。

765 ### seek 和 tell 函数

为了支持随机访问，IO 类型维护一个标记来确定下一个读写操作要在哪里进行。它们还提供了两个函数：一个函数通过将标记 seek 到一个给定位置来重定位它；另一个函数 tell 我们标记的当前位置。标准库实际上定义了两对 seek 和 tell 函数，如表 17.21 所示。一对用于输入流，另一对用于输出流。输入和输出版本的差别在于名字的后缀是 g 还是 p。g 版本表示我们正在“获得”（读取）数据，而 p 版本表示我们正在“放置”（写入）数据。

表 17.21：seek 和 tell 函数

tellg() tellp()	返回一个输入流中（tellg）或输出流中（tellp）标记的当前位置
seekg(pos) seekp(pos)	在一个输入流或输出流中将标记重定位到给定的绝对地址。pos 通常是前一个 tellg 或 tellp 返回的值
seekp(off, from) seekg(off, from)	在一个输入流或输出流中将标记定位到 from 之前或之后 off 个字符，from 可以是下列值之一 • beg，偏移量相对于流开始位置 • cur，偏移量相对于流当前位置 • end，偏移量相对于流结尾位置

从逻辑上讲，我们只能对 istream 和派生自 istream 的类型 ifstream 和 istringstream（参见 8.1 节，第 278 页）使用 g 版本，同样只能对 ostream 和派生自 ostream 的类型 ofstream 和 ostringstream 使用 p 版本。一个 iostream、

fstream 或 stringstream 既能读又能写关联的流，因此对这些类型的对象既能使用 g 版本又能使用 p 版本。

只有一个标记

标准库区分 seek 和 tell 函数的“放置”和“获得”版本这一特性可能会导致误解。即使标准库进行了区分，但它在一个流中只维护单一的标记——并不存在独立的读标记和写标记。

当我们处理一个只读或只写的流时，两种版本的区别甚至是不明显的。我们可以对这些流只使用 g 或只使用 p 版本。如果我们试图对一个 ifstream 流调用 tellp，编译器会报告错误。类似的，编译器也不允许我们对一个 ostringstream 调用 seekg。

fstream 和 stringstream 类型可以读写同一个流。在这些类型中，有单一的缓冲区用于保存读写的数据，同样，标记也只有一个，表示缓冲区中的当前位置。标准库将 g 和 p 版本的读写位置都映射到这个单一的标记。

由于只有单一的标记，因此只要我们在读写操作间切换，就必须进行 seek 操作来重定位标记。 766

重定位标记

seek 函数有两个版本：一个移动到文件中的“绝对”地址；另一个移动到一个给定位置的指定偏移量：

```
// 将标记移动到一个固定位置
seekg(new_position); // 将读标记移动到指定的 pos_type 类型的位置
seekp(new_position); // 将写标记移动到指定的 pos_type 类型的位置

// 移动到给定起始点之前或之后指定的偏移位置
seekg(offset, from); // 将读标记移动到距 from 偏移量为 offset 的位置
seekp(offset, from); // 将写标记移动到距 from 偏移量为 offset 的位置
```

from 的可能值如表 17.21 所示。

参数 new_position 和 offset 的类型分别是 pos_type 和 off_type，这两个类型都是机器相关的，它们定义在头文件 istream 和 ostream 中。pos_type 表示一个文件位置，而 off_type 表示距当前位置的一个偏移量。一个 off_type 类型的值可以是正的也可以是负的，即，我们可以在文件中向前移动或向后移动。

访问标记

函数 tellg 和 tellp 返回一个 pos_type 值，表示流的当前位置。tell 函数通常用来记住一个位置，以便稍后再定位回来：

```
// 记住当前写位置
ostringstream writeStr; // 输出 stringstream
ostringstream::pos_type mark = writeStr.tellp();
// ...
if (cancelEntry)
    // 回到刚才记住的位置
    writeStr.seekp(mark);
```

读写同一个文件

我们来考察一个编程实例。假定已经给定了一个要读取的文件，我们要在此文件的末尾写入新的一行，这一行包含文件中每行的相对起始位置。例如，给定下面文件：

```
abcd
efg
hi
j
```

程序应该生成如下修改过的文件：

767

```
abcd
efg
hi
j
5 9 12 14
```

注意，我们的程序不必输出第一行的偏移——它总是从位置 0 开始。还要注意，统计偏移量时必须包含每行末尾不可见的换行符。最后，注意输出的最后一个数是我们的输出开始那行的偏移量。在输出中包含了这些偏移量后，我们的输出就与文件的原始内容区分开来了。我们可以读取结果文件中最后一个数，定位到对应偏移量，即可得到我们的输出的起始地址。

我们的程序将逐行读取文件。对每一行，我们将递增计数器，将刚刚读取的一行的长度加到计数器上，则此计数器即为下一行的起始地址：

```
int main()
{
    // 以读写方式打开文件，并定位到文件尾
    // 文件模式参数参见 8.2.2 节（第 286 页）
    fstream inOut("copyOut",
                  fstream::ate | fstream::in | fstream::out);
    if (!inOut) {
        cerr << "Unable to open file!" << endl;
        return EXIT_FAILURE; // EXIT_FAILURE 参见 6.3.2 节（第 204 页）
    }
    // inOut 以 ate 模式打开，因此一开始就定义到其文件尾
    auto end_mark = inOut.tellg();     // 记住原文件尾位置
    inOut.seekg(0, fstream::beg);      // 重定位到文件开始
    size_t cnt = 0;                    // 字节数累加器
    string line;                       // 保存输入中的每行
    // 继续读取的条件：还未遇到错误且还在读取原数据
    while (inOut && inOut.tellg() != end_mark
           && getline(inOut, line)) {      // 且还可获取一行输入
        cnt += line.size() + 1;            // 加 1 表示换行符
        auto mark = inOut.tellg();         // 记住读取位置
        inOut.seekp(0, fstream::end);      // 将写标记移动到文件尾
        inOut << cnt;                      // 输出累计的长度
        // 如果不是最后一行，打印一个分隔符
        if (mark != end_mark) inOut << " ";
        inOut.seekg(mark);                 // 恢复读位置
    }
    inOut.seekp(0, fstream::end);          // 定位到文件尾
```

```
        inOut << "\n";                    // 在文件尾输出一个换行符
        return 0;
    }
```

我们的程序用 in、out 和 ate 模式（参见 8.2.2 节，第 286 页）打开 fstream。前两个模式指出我们想读写同一个文件。指定 ate 会将读写标记定位到文件尾。与往常一样，我们检查文件是否成功打开，如果失败就退出（参见 6.3.2 节，第 203 页）。 768

由于我们的程序向输入文件写入数据，因此不能通过文件尾来判断是否停止读取，而是应该在达到原数据的末尾时停止。因此，我们必须首先记住原文件尾的位置。由于我们是以 ate 模式打开文件的，因此 inOut 已经定位到文件尾了。我们将当前位置（即，原文件尾）保存在 end_mark 中。记住文件尾位置之后，我们 seek 到距文件起始位置偏移量为 0 的地方，即，将读标记重定位到文件起始位置。

while 循环的条件由三部分组成：首先检查流是否合法；如果合法，通过比较当前读位置（由 tellg 返回）和记录在 end_mark 中的位置来检查是否读完了原数据；最后，假定前两个检查都已成功，我们调用 getline 读取输入的下一行，如果 getline 成功，则执行 while 循环体。

循环体首先将当前位置记录在 mark 中。我们保存当前位置是为了在输出下一个偏移量后再退回来。接下来调用 seekp 将写标记重定位到文件尾。我们输出计数器的值，然后调用 seekg 回到记录在 mark 中的位置。回退到原位置后，我们就准备好继续检查循环条件了。

每步循环都会输出下一行的偏移量。因此，最后一步循环负责输出最后一行的偏移量。但是，我们还需要在文件尾输出一个换行符。与其他写操作一样，在输出换行符之前我们调用 seekp 来定位到文件尾。

17.5.3 节练习

练习 17.39：对本节给出的 seek 程序，编写你自己的版本。

done!

769

小结

本章介绍了一些特殊 IO 操作和四个标准库类型：tuple、bitset、正则表达式和随机数。

tuple 是一个模板，允许我们将多个不同类型的成员捆绑成单一对象。每个 tuple 包含指定数量的成员，但对一个给定的 tuple 类型，标准库并未限制我们可以定义的成员数量上限。

bitset 允许我们定义指定大小的二进制位集合。标准库不限制一个 bitset 的大小必须与整型类型的大小匹配，bitset 的大小可以更大。除了支持普通的位运算符（参见 4.8 节，第 136 页）外，bitset 还定义了一些命名的操作，允许我们操纵 bitset 中特定位的状态。

正则表达式库提供了一组类和函数：regex 类管理用某种正则表达式语言编写的正则表达式。匹配类保存了某个特定匹配的相关信息。这些类被函数 regex_search 和 regex_match 所用。这两个函数接受一个 regex 对象和一个字符序列，检查 regex 中的正则表达式是否匹配给定的字符序列。regex 迭代器类型是迭代器适配器，它们使用 regex_search 遍历输入序列，返回每个匹配的子序列。标准库还定义了一个 regex_replace 函数，允许我们用指定内容替换输入序列中与正则表达式匹配的部分。

随机数库由一组随机数引擎类和分布类组成。随机数引擎返回一个均匀分布的整型值序列。标准库定义了多个引擎，它们具有不同的性能特点。default_random_engine 是适合于大多数普通情况的引擎。标准库还定义了 20 个分布类型。这些分布类型使用一个引擎来生成指定类型的随机数，这些随机数的值都在给定范围内，且分布满足指定的概率分布。

术语表

bitset 标准库类，保存二进制位集合，大小在编译时已知，并提供检测和设置集合中二进制位的操作。

cmatch csub_match 对象的容器，保存一个 regex 与一个 const char*输入序列匹配的相关信息。容器首元素描述了整个匹配结果。后续元素描述了子表达式的匹配结果。

cregex_iterator 类似 sregex_iterator，唯一的差别是此迭代器遍历一个 char 数组。

csub_match 保存一个正则表达式与一个 const char*匹配结果的类型。可以表示整个匹配或子表达式的匹配。

默认随机数引擎（default random engine） 用于普通用途的随机数引擎的类型别名。

格式化 IO（formatted IO） 读写操作，利用要读写的对象的类型来定义操作的行为。格式化输入操作执行适合要读取的类型的转换操作，如将 ASCII 码字符串转换为算术类型以及（默认地）忽略空白符。格式化输出操作将类型转换为可打印的字符表示形式、补白输出，还可能执行其他与输出类型相关的转换。

770

get 模板函数，返回给定 tuple 的指定成员。例如，get<0>(t)返回 tuplet 的第一个成员。

高位（high-order） bitset 中下标最大的那些位。

低位（low-order） bitset 中下标最小的那些位。

操纵符（manipulator）“操纵”流的类函数对象。操纵符可用作重载的 IO 运算符<<和>>的右侧运算对象。大多数操纵符会改变流对象的内部状态。这种操纵符通常是成对的——一个改变状态，另一个恢复到流的默认状态。

随机数分布（random-number distribution）标准库类型，根据其名字所指出的概率分布转换随机数引擎的输出值。例如，`uniform_int_distribution<T>`生成类型为 `T` 的均匀分布的整数，而 `normal_distribution<T>`生成正态分布的值，依此类推。

随机数引擎（random-number engine）标准库类型，生成随机的无符号数。引擎的设计意图是只用作随机数分布的输入。

随机数发生器（random-number generator）一个随机数引擎类型和一个分布类型的组合。

regex 管理正则表达式的类。

regex_error 异常类型，当正则表达式中存在语法错误时抛出此异常。

regex_match 确定整个输入序列是否与给定 `regex` 对象匹配的函数。

regex_replace 使用一个 `regex` 对象来匹配输入序列并用给定格式替换匹配的子表达式的函数。

regex_search 使用一个 `regex` 对象在给定输入序列中查找匹配的子序列的函数。

正则表达式（regular expression）一种描述字符序列的方式。

种子（seed）提供给随机数引擎的值，使引擎移动到生成的随机数序列中一个新的点。

smatch `ssub_match` 对象的容器，提供一个 `regex` 与一个 `string` 输入序列匹配的相关信息。容器首元素描述了整个匹配结果。后续元素描述了子表达式的匹配结果。

sregex_iterator 迭代器，使用给定的 `regex` 对象遍历一个 `string` 来查找匹配子串。其构造函数通过调用 `regex_search` 将迭代器定位到第一个匹配。递增迭代器的操作会调用 `regex_search`，从给定 `string` 中当前匹配之后的位置开始查找匹配。解引用迭代器返回一个描述当前匹配的 `smatch` 对象。

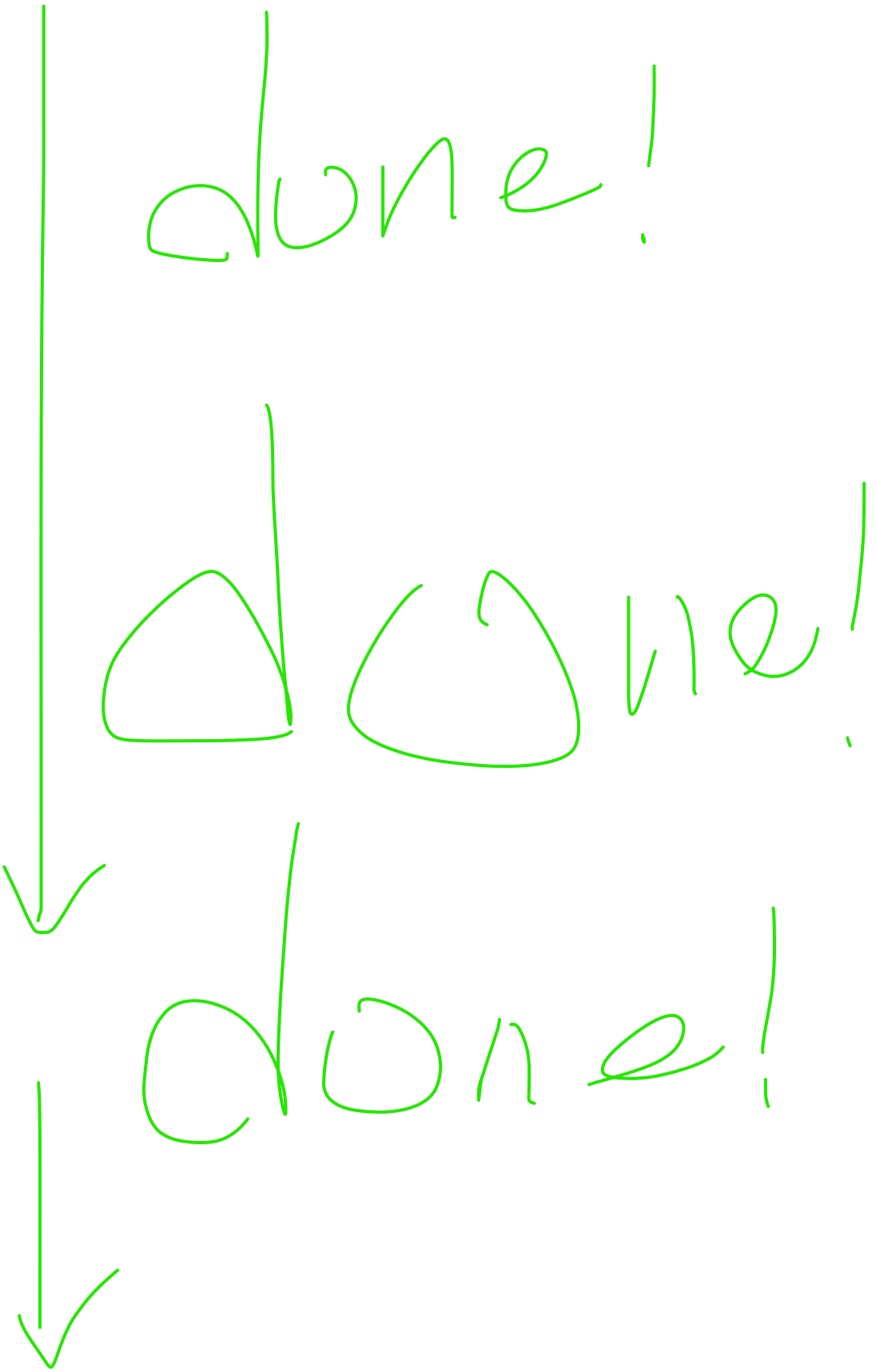
ssub_match 保存正则表达式与 `string` 匹配结果的类型。可以描述整个匹配或子表达式的匹配。

子表达式（subexpression）正则表达式模式中用括号包围的组成部分。

tuple 模板，生成的类型保存指定类型的未命名成员。标准库没有限制一个 `tuple` 最多可以包含多少个成员。

未格式化 IO（unformatted IO）将流当作无差别的字节流来处理的操作。未格式化操作给用户增加了很多管理 IO 的负担。

done!


done!
done!
done!

第 18 章
用于大型程序的工具

内容

C++语言能解决的问题规模千变万化，有的小到一个程序员几小时就能完成，有的则是含有几千几万行代码的庞大系统，需要几百个程序员协同工作好几年。本书之前介绍的内容对各种规模的编程问题都适用。

除此之外，C++语言还包含其他一些特征，当我们编写比较复杂的、小组和个人难以管理的系统时，这些特征最为有用。本章的主题即是向读者介绍这些特征，它们包括异常处理、命名空间和多重继承。

772 与仅需几个程序员就能开发完成的系统相比，大规模编程对程序设计语言的要求更高。大规模应用程序的特殊要求包括：

- 在独立开发的子系统之间协同处理错误的能力。
- 使用各种库（可能包含独立开发的库）进行协同开发的能力。
- 对比较复杂的应用概念建模的能力。

本章介绍的三种 C++语言特性正好能满足上述要求，它们是：异常处理、命名空间和多重继承。

18.1 异常处理

异常处理（exception handling）机制允许程序中独立开发的部分能够在运行时就出现的问题进行通信并做出相应的处理。异常使得我们能够将问题的检测与解决过程分离开来。程序的一部分负责检测问题的出现，然后解决该问题的任务传递给程序的另一部分。检测环节无须知道问题处理模块的所有细节，反之亦然。

在 5.6 节（第 173 页）我们曾介绍过一些有关异常处理的基本概念和机理，本节将继续扩展这些知识。对于程序员来说，要想有效地使用异常处理，必须首先了解当抛出异常时发生了什么，捕获异常时发生了什么，以及用来传递错误的对象的意义。

18.1.1 抛出异常

在 C++语言中，我们通过**抛出**（throwing）一条表达式来**引发**（raised）一个异常。被抛出的表达式的类型以及当前的调用链共同决定了哪段**处理代码**（handler）将被用来处理该异常。被选中的处理代码是在调用链中与抛出对象类型匹配的最近的处理代码。其中，根据抛出对象的类型和内容，程序的异常抛出部分将会告知异常处理部分到底发生了什么错误。

当执行一个 `throw` 时，跟在 `throw` 后面的语句将不再被执行。相反，程序的控制权从 `throw` 转移到与之匹配的 `catch` 模块。该 `catch` 可能是同一个函数中的局部 `catch`，也可能位于直接或间接调用了发生异常的函数的另一个函数中。控制权从一处转移到另一处，这有两个重要的含义：

- 沿着调用链的函数可能会提早退出。
- 一旦程序开始执行异常处理代码，则沿着调用链创建的对象将被销毁。

因为跟在 `throw` 后面的语句将不再被执行，所以 `throw` 语句的用法有点类似于 `return` 语句：它通常作为条件语句的一部分或者作为某个函数的最后（或者唯一）一条语句。

773 **栈展开**

当抛出一个异常后，程序暂停当前函数的执行过程并立即开始寻找与异常匹配的 `catch` 子句。当 `throw` 出现在一个 **`try` 语句块**（try block）内时，检查与该 `try` 块关联的 `catch` 子句。如果找到了匹配的 `catch`，就使用该 `catch` 处理异常。如果这一步没找到匹配的 `catch` 且该 `try` 语句嵌套在其他 `try` 块中，则继续检查与外层 `try` 匹配的 `catch` 子句。如果还是找不到匹配的 `catch`，则退出当前的函数，在调用当前函数的外层函数中继续寻找。

如果对抛出异常的函数的调用语句位于一个 `try` 语句块内，则检查与该 `try` 块关联

的 catch 子句。如果找到了匹配的 catch，就使用该 catch 处理异常。否则，如果该 try 语句嵌套在其他 try 块中，则继续检查与外层 try 匹配的 catch 子句。如果仍然没有找到匹配的 catch，则退出当前这个主调函数，继续在调用了刚刚退出的这个函数的其他函数中寻找，以此类推。

上述过程被称为**栈展开**（stack unwinding）过程。栈展开过程沿着嵌套函数的调用链不断查找，直到找到了与异常匹配的 catch 子句为止；或者也可能一直没找到匹配的 catch，则退出主函数后查找过程终止。

假设找到了一个匹配的 catch 子句，则程序进入该子句并执行其中的代码。当执行完这个 catch 子句后，找到与 try 块关联的最后一个 catch 子句之后的点，并从这里继续执行。

如果没找到匹配的 catch 子句，程序将退出。因为异常通常被认为是妨碍程序正常执行的事件，所以一旦引发了某个异常，就不能对它置之不理。当找不到匹配的 catch 时，程序将调用标准库函数 **terminate**，顾名思义，terminate 负责终止程序的执行过程。

一个异常如果没有被捕获，则它将终止当前的程序。

栈展开过程中对象被自动销毁

在栈展开过程中，位于调用链上的语句块可能会提前退出。通常情况下，程序在这些块中创建了一些局部对象。我们已经知道，块退出后它的局部对象也将随之销毁，这条规则对于栈展开过程同样适用。如果在栈展开过程中退出了某个块，编译器将负责确保在这个块中创建的对象能被正确地销毁。如果某个局部对象的类型是类类型，则该对象的析构函数将被自动调用。与往常一样，编译器在销毁内置类型的对象时不需要做任何事情。

如果异常发生在构造函数中，则当前的对象可能只构造了一部分。有的成员已经初始化了，而另外一些成员在异常发生前也许还没有初始化。即使某个对象只构造了一部分，我们也要确保已构造的成员能被正确地销毁。

类似的，异常也可能发生在数组或标准库容器的元素初始化过程中。与之前类似，如果在异常发生前已经构造了一部分元素，则我们应该确保这部分元素被正确地销毁。

析构函数与异常

774

析构函数总是会被执行的，但是函数中负责释放资源的代码却可能被跳过，这一特点对于我们如何组织程序结构有重要影响。如我们在 12.1.4 节（第 415 页）介绍过的，如果一个块分配了资源，并且在负责释放这些资源的代码前面发生了异常，则释放资源的代码将不会被执行。另一方面，类对象分配的资源将由类的析构函数负责释放。因此，如果我们使用类来控制资源的分配，就能确保无论函数正常结束还是遭遇异常，资源都能被正确地释放。

析构函数在栈展开的过程中执行，这一事实影响着我们编写析构函数的方式。在栈展开的过程中，已经引发了异常但是我们还没有处理它。如果异常抛出后没有被正确捕获，则系统将调用 terminate 函数。因此，出于栈展开可能使用析构函数的考虑，析构函数不应该抛出不能被它自身处理的异常。换句话说，如果析构函数需要执行某个可能抛出异常的操作，则该操作应该被放置在一个 try 语句块当中，并且在析构函数内部得到处理。

在实际的编程过程中，因为析构函数仅仅是释放资源，所以它不太可能抛出异常。所有标准库类型都能确保它们的析构函数不会引发异常。

在栈展开的过程中，运行类类型的局部对象的析构函数。因为这些析构函数是自动执行的，所以它们不应该抛出异常。一旦在栈展开的过程中析构函数抛出了异常，并且析构函数自身没能捕获到该异常，则程序将被终止。

异常对象

异常对象（exception object）是一种特殊的对象，编译器使用异常抛出表达式来对异常对象进行拷贝初始化（参见 13.1.1 节，第 441 页）。因此，throw 语句中的表达式必须拥有完全类型（参见 7.3.3 节，第 250 页）。而且如果该表达式是类类型的话，则相应的类必须含有一个可访问的析构函数和一个可访问的拷贝或移动构造函数。如果该表达式是数组类型或函数类型，则表达式将被转换成与之对应的指针类型。

异常对象位于由编译器管理的空间中，编译器确保无论最终调用的是哪个 catch 子句都能访问该空间。当异常处理完毕后，异常对象被销毁。

如我们所知，当一个异常被抛出时，沿着调用链的块将依次退出直至找到与异常匹配的处理代码。如果退出了某个块，则同时释放块中局部对象使用的内存。因此，抛出一个指向局部对象的指针几乎肯定是一种错误的行为。出于同样的原因，从函数中返回指向局部对象的指针也是错误的（参见 6.3.2 节，第 202 页）。如果指针所指的对象位于某个块中，而该块在 catch 语句之前就已经退出了，则意味着在执行 catch 语句之前局部对象已经被销毁了。

当我们抛出一条表达式时，该表达式的静态编译时类型（参见 15.2.3 节，第 534 页）决定了异常对象的类型。读者必须牢记这一点，因为很多情况下程序抛出的表达式类型来
775
自于某个继承体系。如果一条 throw 表达式解引用一个基类指针，而该指针实际指向的是派生类对象，则抛出的对象将被切掉一部分（参见 15.2.3 节，第 535 页），只有基类部分被抛出。

抛出指针要求在任何对应的处理代码存在的地方，指针所指的对象都必须存在。

18.1.1 节练习

练习 18.1：在下列 throw 语句中异常对象的类型是什么？

(a)
```
range_error r("error");
throw r;
```

(b)
```
exception *p = &r;
throw *p;
```

如果将（b）中的 throw 语句写成了 throw p 将发生什么情况？

练习 18.2：当在指定的位置发生了异常时将出现什么情况？

```
void exercise(int *b, int *e)
{
    vector<int> v(b, e);
    int *p = new int[v.size()];
    ifstream in("ints");
    // 此处发生异常
}
```

练习 18.3：要想让上面的代码在发生异常时能正常工作，有两种解决方案。请描述这两种方法并实现它们。

18.1.2 捕获异常

catch 子句（catch clause）中的**异常声明**（exception declaration）看起来像是只包含一个形参的函数形参列表。像在形参列表中一样，如果 catch 无须访问抛出的表达式的话，则我们可以忽略捕获形参的名字。

声明的类型决定了处理代码所能捕获的异常类型。这个类型必须是完全类型（参见 7.3.3 节，第 250 页），它可以是左值引用，但不能是右值引用（参见 13.6.1 节，第 471 页）。

当进入一个 catch 语句后，通过异常对象初始化异常声明中的参数。和函数的参数类似，如果 catch 的参数类型是非引用类型，则该参数是异常对象的一个副本，在 catch 语句内改变该参数实际上改变的是局部副本而非异常对象本身；相反，如果参数是引用类型，则和其他引用参数一样，该参数是异常对象的一个别名，此时改变参数也就是改变异常对象。

catch 的参数还有一个特性也与函数的参数非常类似：如果 catch 的参数是基类类型，则我们可以使用其派生类类型的异常对象对其进行初始化。此时，如果 catch 的参数是非引用类型，则异常对象将被切掉一部分（参见 15.2.3 节，第 535 页），这与将派生 776 类对象以值传递的方式传给一个普通函数差不多。另一方面，如果 catch 的参数是基类的引用，则该参数将以常规方式绑定到异常对象上。

最后一点需要注意的是，异常声明的静态类型将决定 catch 语句所能执行的操作。如果 catch 的参数是基类类型，则 catch 无法使用派生类特有的任何成员。

Best Practices 通常情况下，如果 catch 接受的异常与某个继承体系有关，则最好将该 catch 的参数定义成引用类型。

查找匹配的处理代码

在搜寻 catch 语句的过程中，我们最终找到的 catch 未必是异常的最佳匹配。相反，挑选出来的应该是第一个与异常匹配的 catch 语句。因此，越是专门的 catch 越应该置于整个 catch 列表的前端。

因为 catch 语句是按照其出现的顺序逐一进行匹配的，所以当程序使用具有继承关系的多个异常时必须对 catch 语句的顺序进行组织和管理，使得派生类异常的处理代码出现在基类异常的处理代码之前。

与实参和形参的匹配规则相比，异常和 catch 异常声明的匹配规则受到更多限制。此时，绝大多数类型转换都不被允许，除了一些极细小的差别之外，要求异常的类型和 catch 声明的类型是精确匹配的：

- 允许从非常量向常量的类型转换，也就是说，一条非常量对象的 throw 语句可以匹配一个接受常量引用的 catch 语句。
- 允许从派生类向基类的类型转换。
- 数组被转换成指向数组（元素）类型的指针，函数被转换成指向该函数类型的指针。

除此之外，包括标准算术类型转换和类类型转换在内，其他所有转换规则都不能在匹配

catch 的过程中使用。

如果在多个 catch 语句的类型之间存在着继承关系，则我们应该把继承链最底端的类（most derived type）放在前面，而将继承链最顶端的类（least derived type）放在后面。

重新抛出

有时，一个单独的 catch 语句不能完整地处理某个异常。在执行了某些校正操作之后，当前的 catch 可能会决定由调用链更上一层的函数接着处理异常。一条 catch 语句通过**重新抛出**（rethrowing）的操作将异常传递给另外一个 catch 语句。这里的重新抛出仍然是一条 throw 语句，只不过不包含任何表达式：

```
throw;
```

777 空的 throw 语句只能出现在 catch 语句或 catch 语句直接或间接调用的函数之内。如果在处理代码之外的区域遇到了空 throw 语句，编译器将调用 terminate。

一个重新抛出语句并不指定新的表达式，而是将当前的异常对象沿着调用链向上传递。

很多时候，catch 语句会改变其参数的内容。如果在改变了参数的内容后 catch 语句重新抛出异常，则只有当 catch 异常声明是引用类型时我们对参数所做的改变才会被保留并继续传播：

```
catch (my_error &eObj) {          // 引用类型
    eObj.status = errCodes::severeErr;    // 修改了异常对象
    throw;                        // 异常对象的 status 成员是 severeErr
} catch (other_error eObj) {              // 非引用类型
    eObj.status = errCodes::badErr;       // 只修改了异常对象的局部副本
    throw;       // 异常对象的 status 成员没有改变
}
```

捕获所有异常的处理代码

有时我们希望不论抛出的异常是什么类型，程序都能统一捕获它们。要想捕获所有可能的异常是比较有难度的，毕竟有些情况下我们也不知道异常的类型到底是什么。即使我们知道所有的异常类型，也很难为所有类型提供唯一一个 catch 语句。为了一次性捕获所有异常，我们使用省略号作为异常声明，这样的处理代码称为**捕获所有异常**（catch-all）的处理代码，形如 catch(...)。一条捕获所有异常的语句可以与任意类型的异常匹配。

catch(...)通常与重新抛出语句一起使用，其中 catch 执行当前局部能完成的工作，随后重新抛出异常：

```
void manip() {
    try {
        // 这里的操作将引发并抛出一个异常
    }
    catch (...) {
        // 处理异常的某些特殊操作
        throw;
    }
}
```