第17章 特殊库特殊设施

17.1 tuple 类型

1. 我们可以将 tuple 看作一个“快速而随意”的数据结构。可以有任意数量的成员，且成员的类型不受限制。

17.1.1 定义和初始化 tuple

1. 当我们创建一个 tuple 时，需要指出每个成员的类型。
2. 创建一个 tuple 对象时，可以使用它的默认构造函数，它会对每个成员进行值初始化。
3. 当使用初始值初始化对象时，构造函数是 explicit 。因此必须使用直接初始化语法：

```
tuple<size_t, int, double> item = {1, 2, 3.0}; // 错误
tuple<size_t, int, double> item(1, 2, 3.0);    // 正确

// 如果不知道tuple准确的类型细节信息，
// 可以用两个辅助类模板来查询tuple成员的数量和类型
typedef decltype(item) trans;// trans是item的类型

// 返回trans类型对象中成员的数量：
size_t sz = tuple_size<trans>::value;   // 返回3
// cnt的类型与item中第二个成员相同
tuple_element<1, trans>::type cnt = get<1>(item);   // cnt是一个int
```

4. 由于 tuple 定义了 < 和 == 运算符，我们可以将 tuple 序列传递给算法，并且可以在无序容器中将 tuple 作为关键字类型。

17.2 bitset 类型

17.2.1 定义和初始化 bitset

1. 定义一个 bitset 需要声明它包含多好个二进制位。
2. string 的下标编号习惯与 bitset 恰好相反： string 中下标最大的字符（最右字符）用来初始化 bitset 中的低位（下标为0的二进制位）。

17.2.2 bitset 操作

1. 编写一个程序表示30给学生的测试结果-“通过/失败”

```
bool status;

// 使用位运算符的版本
unsigned long quizA = 0;      // 此值被当作位集合使用
quizA |= 1UL << 27;           // 指出第27个学生通过了测试
status = quizA & (1UL << 27);// 检查第27个学生是否通过了测验
quizA &= ~(1UL << 27);        // 第27个学生未通过测验

// 使用标准库bitset完成等价的工作
bitset<30> quizB;             // 每个学生分配一位，所有位都被初始化为0
quizB.set(27);                // 指出第27个学生通过了测试
status = quizB[27];           // 检查第27个学生是否通过了测验
quizB.reset(27);              // 第27个学生未通过测验
```

17.3 正则表达式

1. **正则表达式（regular expression）** 是一种描述符号序列的方法，是一种极其强大的计算工具。

17.3.1 使用正则表达式库

1. 使用正则表达式来查找违反拼写规则“i除非在c之后，否则必须在e之前”单词的例子：

```
// 查找不在字符c之后的字符串ei
string pattern("[^c]ei");
// 我们需要包含pattern的整个单词
pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*";
regex r(pattern);   // 构造一个用于查找模式的regex
smatch results;     // 定义一个对象保存搜索结果
// 定义一个string保存与模式匹配和不匹配的文本
string test_str = "receipt freind theif receive";
// 用r在test_str中查找与pattern匹配的子串
if(regex_search(test_str, results, r))  // 如果有匹配字串
    cout << results.str() << endl;      // 打印匹配的单词
```

2. 一个正则表达式所表示的“程序”是在运行时而非编译时编译的。表达式的编译是一个很慢的操作，特别是在使用了扩展的或者复杂的语法时。因此应该避免创建不必要的 regex 正则表达式。
3. 我们可以搜索多种类型的输入序列，重点在于我们使用的RE库类型必须与输入序列类型匹配：

```
regex r("[[:alnum:]] +\\.(cpp|cxx|cc)$", regex::icase);

// 将匹配string输入序列，而不是char*
smatch results;
if(regex_search("myfile.cc", result, r)) // 错误：输入为char*
    cout << results.str() << endl;

// 将匹配字符数组输入序列
cmatch results;
if(regex_search("myfile.cc", result, r))
    cout << results.str() << endl;       // 打印当前匹配
```

17.3.2 匹配与 Regex 迭代器类型

1. 扩展之前的程序，使用 sregex_iterator 来进行搜索。

```
// 查找前一个不在字符c之后的字符串ei
string pattern("[^c]ei");
// 我们需要包含pattern的整个单词
pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*";
regex r(pattern, regex::icase);   // 进行匹配时将忽略大小写
// 它将返回利用regex_search来寻找文件中的所有匹配
for(sregex_iterator it(file.begin(), file.end(), r), end_it;
        it != end_it; ++it)
        cout << it->str() << endl;  // 匹配的单词
```

17.3.3 使用子表达式

1. 正则表达式中的模式通常包含一个或多个**子表达式（subexpression）**。一个子表达式是模式的一部分，本身也具有意义。正则表达式语法通常用括号表示子表达式。

17.3.4 使用 regex_replace

1. 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。
2. 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。

17.4 随机数

17.4.1 随机数引擎和分布

1. 使用随机数引擎生成10个随机数：

```
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() << " ";
```

2. 为了得到一个指定范围内的数，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;    // 生成随机无符号数
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) << " ";

// 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。
```

3. 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。
4. 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。
5. 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。
6. 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。引擎可以利用它从序列中一个新的位置重新开始生成随机数。
7. 选择一个好的种子，与生成好的随机数所涉及到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` 。
8. 如果程序作为一个自动过程的一部分返回运行，将 `time` 的返回值作为种子的方法就失效了；它可能多次使用的都是相同的种子。

17.4.2 其他随机数分布

1. 由于引擎返回相同的随机数序列，所以我们必须在循环外声明引擎对象。否则，每次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。

17.5 IO 库再探

17.5.1 格式化输入和输出

1. 除了条件状态外，每一个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、浮点数的精度、一个输出元素的宽度等。
2. 标准库定义了一组**操纵符(manipulator)**来修改流的格式状态。
3. 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。
4. 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。
5. 默认情况下，整型值的输入输出使用十进制。我们可以使用操作符 `hex` 、 `oct` 和 `dec` 将其改为十六进制、八进制或者改回十进制。这些操纵符只影响整型运算对象，浮点值的表示形式不受影响。
6. 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。 `noshowbase` 恢复状态，不再显示整型值进制。
7. 默认情况下，精度会控制打印的数字总数，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。
8. 除非你需要控制浮点数的表达形式（如，按列打印数据或打印表示金额或百分比的数据），否则由标准库选择计数法是最好的方式。
9. 通过使用恰当的操作符，我们可以强制一个流使用科学计数法(scientific)、定点十进制(fixed)或是十六进制(hexfloat)计数法。

17.5.2 未格式化的输入/输出操作

1. 标准库提供了一组**底层操作**，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。
2. `get` 将分隔符留作 `istream` 中下一个字符，而 `getline` 则读取并丢弃分隔符。
3. 一个常见的错误是本想从流中删除分隔符，但却忘了做。

17.5.3 流随机访问

1. 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，以此类推。

2. 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)我们当前的位置。
3. 由于 `istream` 和 `ostream` 类型通常不支持随机访问，所以随机访问功能通常适用于 `fstream` 和 `stream` 类型。