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。

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。

第 18 章
用于大型程序的工具

内容

18.1 异常处理 684
18.2 命名空间 695
18.3 多重继承与虚继承 710
小结 722
术语表 722

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。

776

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;
    }
}
```

`catch(...)`既能单独出现，也能与其他几个 `catch` 语句一起出现。



如果 `catch(...)` 与其他几个 `catch` 语句一起出现，则 `catch(...)` 必须在最后的位置。出现在捕获所有异常语句后面的 `catch` 语句将永远不会被匹配。

18.1.2 节练习

练习 18.4：查看图 18.1（第 693 页）所示的继承体系，说明下面的 `try` 块有何错误并修改它。

```
try {
    // 使用 C++标准库
} catch(exception) {
    // ...
} catch(const runtime_error &re) {
    // ...
} catch(overflow_error eobj) { /* ... */ }
```

练习 18.5：修改下面的 `main` 函数，使其能捕获图 18.1（第 693 页）所示的任何异常类型：

```
int main() {
    // 使用 C++标准库
}
```

处理代码应该首先打印异常相关的错误信息，然后调用 `abort`（定义在 `cstdlib` 头文件中）终止 `main` 函数。

练习 18.6：已知下面的异常类型和 `catch` 语句，书写一个 `throw` 表达式使其创建的异常对象能被这些 `catch` 语句捕获：

```
(a) class exceptionType { };
    catch(exceptionType *pet) { }
(b) catch(...) { }
(c) typedef int EXCPTYPE;
    catch(EXCPTYPE) { }
```

18.1.3 函数 try 语句块与构造函数

通常情况下，程序执行的任何时刻都可能发生异常，特别是异常可能发生在处理构造函数初始值的过程中。构造函数在进入其函数体之前首先执行初始值列表。因为在初始值列表抛出异常时构造函数体内的 `try` 语句块还未生效，所以构造函数体内的 `catch` 语句无法处理构造函数初始值列表抛出的异常。

778

要想处理构造函数初始值抛出的异常，我们必须将构造函数写成**函数 try 语句块**（也称为函数测试块，function try block）的形式。函数 `try` 语句块使得一组 `catch` 语句既能处理构造函数体（或析构函数体），也能处理构造函数的初始化过程（或析构函数的析构过程）。举个例子，我们可以把 `Blob` 的构造函数（参见 16.1.2 节，第 586 页）置于一个函数 `try` 语句块中：

```
template <typename T>
Blob<T>::Blob(std::initializer_list<T> il) try :
```

```
        data(std::make_shared<std::vector<T>>(il)) {
    /* 空函数体*/
} catch(const std::bad_alloc &e) { handle_out_of_memory(e); }
```

注意：关键字 try 出现在表示构造函数初始值列表的冒号以及表示构造函数体（此例为空）的花括号之前。与这个 try 关联的 catch 既能处理构造函数体抛出的异常，也能处理成员初始化列表抛出的异常。

779

还有一种情况值得读者注意，在初始化构造函数的参数时也可能发生异常，这样的异常不属于函数 try 语句块的一部分。函数 try 语句块只能处理构造函数开始执行后发生的异常。和其他函数调用一样，如果在参数初始化的过程中发生了异常，则该异常属于调用表达式的一部分，并将在调用者所在的上下文中处理。



处理构造函数初始值异常的唯一方法是将构造函数写成函数 try 语句块。

18.1.3 节练习

练习 18.7：根据第 16 章的介绍定义你自己的 Blob 和 BlobPtr，注意将构造函数写成函数 try 语句块。

18.1.4 noexcept 异常说明

对于用户及编译器来说，预先知道某个函数不会抛出异常显然大有裨益。首先，知道函数不会抛出异常有助于简化调用该函数的代码；其次，如果编译器确认函数不会抛出异常，它就能执行某些特殊的优化操作，而这些优化操作并不适用于可能出错的代码。

C++ 11

在 C++11 新标准中，我们可以通过提供 **noexcept 说明**（noexcept specification）指定某个函数不会抛出异常。其形式是关键字 noexcept 紧跟在函数的参数列表后面，用以标识该函数不会抛出异常：

```
void recoup(int) noexcept;          // 不会抛出异常
void alloc(int);                    // 可能抛出异常
```

这两条声明语句指出 recoup 将不会抛出任何异常，而 alloc 可能抛出异常。我们说 recoup 做了**不抛出说明**（nonthrowing specification）。

对于一个函数来说，noexcept 说明要么出现在该函数的所有声明语句和定义语句中，要么一次也不出现。该说明应该在函数的尾置返回类型（参见 6.3.3 节，第 206 页）之前。我们也可以在函数指针的声明和定义中指定 noexcept。在 typedef 或类型别名中则不能出现 noexcept。在成员函数中，noexcept 说明符需要跟在 const 及引用限定符之后，而在 final、override 或虚函数的=0 之前。

违反异常说明

读者需要清楚的一个事实是编译器并不会在编译时检查 noexcept 说明。实际上，如果一个函数在说明了 noexcept 的同时又含有 throw 语句或者调用了可能抛出异常的其他函数，编译器将顺利编译通过，并不会因为这种违反异常说明的情况而报错（不排除个别编译器会对这种用法提出警告）：

780

```
// 尽管该函数明显违反了异常说明，但它仍然可以顺利编译通过
void f() noexcept                   // 承诺不会抛出异常
{
```

```
    throw exception();          // 违反了异常说明
}
```

因此可能出现这样一种情况：尽管函数声明了它不会抛出异常，但实际上还是抛出了。一旦一个 noexcept 函数抛出了异常，程序就会调用 terminate 以确保遵守不在运行时抛出异常的承诺。上述过程对是否执行栈展开未作约定，因此 noexcept 可以用在两种情况下：一是我们确认函数不会抛出异常，二是我们根本不知道该如何处理异常。

指明某个函数不会抛出异常可以令该函数的调用者不必再考虑如何处理异常。无论是函数确实不抛出异常，还是程序被终止，调用者都无须为此负责。



通常情况下，编译器不能也不必在编译时验证异常说明。

向后兼容：异常说明

早期的 C++版本设计了一套更加详细的异常说明方案，该方案使得我们可以指定某个函数可能抛出的异常类型。函数可以指定一个关键字 throw，在后面跟上括号括起来的异常类型列表。throw 说明符所在的位置与新版本 C++中 noexcept 所在的位置相同。

上述使用 throw 的异常说明方案在 C++11 新版本中已经被取消了。然而尽管如此，它还有一个重要的用处。如果函数被设计为是 throw()的，则意味着该函数将不会抛出异常：

```
void recoup(int) noexcept;      // recoup 不会抛出异常
void recoup(int) throw();       // 等价的声明
```

上面的两条声明语句是等价的，它们都承诺 recoup 不会抛出异常。

异常说明的实参

noexcept 说明符接受一个可选的实参，该实参必须能转换为 bool 类型：如果实参是 true，则函数不会抛出异常；如果实参是 false，则函数可能抛出异常：

```
void recoup(int) noexcept(true);    // recoup 不会抛出异常
void alloc(int) noexcept(false);    // alloc 可能抛出异常
```

noexcept 运算符

noexcept 说明符的实参常常与 **noexcept 运算符**（noexcept operator）混合使用。noexcept 运算符是一个一元运算符，它的返回值是一个 bool 类型的右值常量表达式，用于表示给定的表达式是否会抛出异常。和 sizeof（参见 4.9 节，第 139 页）类似，noexcept 也不会求其运算对象的值。

C++ 11

781

例如，因为我们声明 recoup 时使用了 noexcept 说明符，所以下面的表达式的返回值为 true：

```
noexcept(recoup(i))  // 如果 recoup 不抛出异常则结果为 true；否则结果为 false
```

更普通的形式是：

```
noexcept(e)
```

当 e 调用的所有函数都做了不抛出说明且 e 本身不含有 throw 语句时，上述表达式为 true；否则 noexcept(e)返回 false。

我们可以使用 noexcept 运算符得到如下的异常说明：

```
void f() noexcept(noexcept(g())); // f 和 g 的异常说明一致
```

如果函数 g 承诺了不会抛出异常，则 f 也不会抛出异常；如果 g 没有异常说明符，或者 g 虽然有异常说明符但是允许抛出异常，则 f 也可能抛出异常。



noexcept 有两层含义：当跟在函数参数列表后面时它是异常说明符；而当作为 noexcept 异常说明的 bool 实参出现时，它是一个运算符。

异常说明与指针、虚函数和拷贝控制

尽管 noexcept 说明符不属于函数类型的一部分，但是函数的异常说明仍然会影响函数的使用。

函数指针及该指针所指的函数必须具有一致的异常说明。也就是说，如果我们为某个指针做了不抛出异常的声明，则该指针将只能指向不抛出异常的函数。相反，如果我们显式或隐式地说明了指针可能抛出异常，则该指针可以指向任何函数，即使是承诺了不抛出异常的函数也可以：

```
// recoup 和 pf1 都承诺不会抛出异常
void (*pf1)(int) noexcept = recoup;
// 正确：recoup 不会抛出异常，pf2 可能抛出异常，二者之间互不干扰
void (*pf2)(int) = recoup;

pf1 = alloc;      // 错误：alloc 可能抛出异常，但是 pf1 已经说明了它不会抛出异常
pf2 = alloc;      // 正确：pf2 和 alloc 都可能抛出异常
```

如果一个虚函数承诺了它不会抛出异常，则后续派生出来的虚函数也必须做出同样的承诺；与之相反，如果基类的虚函数允许抛出异常，则派生类的对应函数既可以允许抛出异常，也可以不允许抛出异常：

```
class Base {
public:
    virtual double f1(double) noexcept;   // 不会抛出异常
    virtual int f2() noexcept(false);     // 可能抛出异常
    virtual void f3();                    // 可能抛出异常
};
```

782

```
class Derived : public Base {
public:
    double f1(double);             // 错误：Base::f1 承诺不会抛出异常
    int f2() noexcept(false);      // 正确：与 Base::f2 的异常说明一致
    void f3() noexcept;            // 正确：Derived 的 f3 做了更严格的限定，
                                   // 这是允许的
};
```

当编译器合成拷贝控制成员时，同时也生成一个异常说明。如果对所有成员和基类的所有操作都承诺了不会抛出异常，则合成的成员是 noexcept 的。如果合成成员调用的任意一个函数可能抛出异常，则合成的成员是 noexcept(false)。而且，如果我们定义了一个析构函数但是没有为它提供异常说明，则编译器将合成一个。合成的异常说明将与假设由编译器为类合成析构函数时所得的异常说明一致。

18.1.4 节练习

练习 18.8：回顾你之前编写的各个类，为它们的构造函数和析构函数添加正确的异常说明。如果你认为某个析构函数可能抛出异常，尝试修改代码使得该析构函数不会抛出异常。

18.1.5 异常类层次

标准库异常类（参见 5.6.3 节，第 176 页）构成了图 18.1 所示的继承体系（参见第 15 章）。

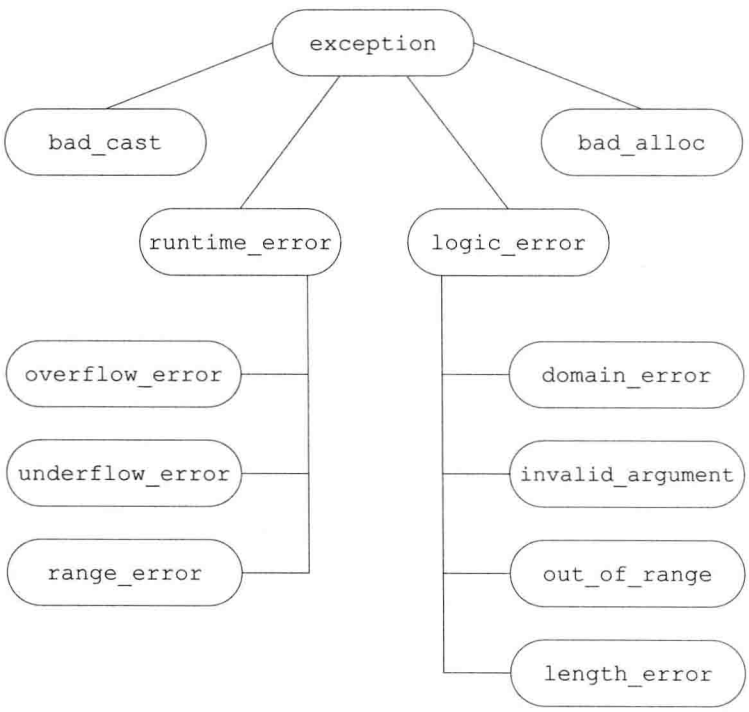


图 18.1：标准 exception 类层次

类型 exception 仅仅定义了拷贝构造函数、拷贝赋值运算符、一个虚析构函数和一个名为 what 的虚成员。其中 what 函数返回一个 const char*，该指针指向一个以 unll 结尾的字符数组，并且确保不会抛出任何异常。

类 exception、bad_cast 和 bad_alloc 定义了默认构造函数。类 runtime_error 和 logic_error 没有默认构造函数，但是有一个可以接受 C 风格字符串或者标准库 string 类型实参的构造函数，这些实参负责提供关于错误的更多信息。在这些类中，what 负责返回用于初始化异常对象的信息。因为 what 是虚函数，所以当我们捕获基类的引用时，对 what 函数的调用将执行与异常对象动态类型对应的版本。

书店应用程序的异常类

实际的应用程序通常会自定义 exception（或者 exception 的标准库派生类）的派生类以扩展其继承体系。这些面向应用的异常类表示了与应用相关的异常条件。

如果我们构建的是一个真实的书店应用程序，则其中的类将比本书之前所示的复杂得多。复杂性的一个方面就是如何处理异常。实际上，我们很可能需要建立一个自己的异常

783

类体系，用它来表示与应用相关的各种问题。我们设计的异常类可能如下所示：

```
// 为某个书店应用程序设定的异常类
class out_of_stock: public std::runtime_error {
public:
    explicit out_of_stock(const std::string &s):
                        std::runtime_error(s) { }
};
class isbn_mismatch: public std::logic_error {
public:
    explicit isbn_mismatch(const std::string &s):
                            std::logic_error(s) { }
    isbn_mismatch(const std::string &s,
        const std::string &lhs, const std::string &rhs):
        std::logic_error(s), left(lhs), right(rhs) { }
    const std::string left, right;
};
```

784 由上可知，我们的面向应用的异常类继承自标准异常类。和其他继承体系一样，异常类也可以看作按照层次关系组织的。层次越低，表示的异常情况就越特殊。例如，在异常类继承体系中位于最顶层的通常是 exception，exception 表示的含义是某处出错了，至于错误的细节则未作描述。

继承体系的第二层将 exception 划分为两个大的类别：运行时错误和逻辑错误。运行时错误表示的是只有在程序运行时才能检测到的错误；而逻辑错误一般指的是我们可以在程序代码中发现的错误。

我们的书店应用程序进一步细分上述异常类别。名为 out_of_stock 的类表示在运行时可能发生的错误，比如某些顺序无法满足；名为 isbn_mismatch 的类表示 logic_error 的一个特例，程序可以通过比较对象的 isbn()结果来阻止或处理这一错误。

使用我们自己的异常类型

我们使用自定义异常类的方式与使用标准异常类的方式完全一样。程序在某处抛出异常类型的对象，在另外的地方捕获并处理这些出现的问题。举个例子，我们可以为 Sales_data 类定义一个复合加法运算符，当检测到参与加法的两个 ISBN 编号不一致时抛出名为 isbn_mismatch 的异常：

```
// 如果参与加法的两个对象并非同一书籍，则抛出一个异常
Sales_data&
Sales_data::operator+=(const Sales_data& rhs)
{
    if (isbn() != rhs.isbn())
        throw isbn_mismatch("wrong isbns", isbn(), rhs.isbn());
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}
```

使用了复合加法运算符的代码将能检测到这一错误，进而输出一条相应的错误信息并继续完成其他任务：

```
// 使用之前设定的书店程序异常类
Sales_data item1, item2, sum;
while (cin >> item1 >> item2) {          // 读取两条交易信息
    try {
        sum = item1 + item2;             // 计算它们的和
        // 此处使用 sum
    } catch (const isbn_mismatch &e) {
      cerr << e.what() << ": left isbn(" << e.left
           << ") right isbn(" << e.right << ")" << endl;
    }
}
```

18.1.5 节练习

785

练习 18.9：定义本节描述的书店程序异常类，然后为 Sales_data 类重新编写一个复合赋值运算符并令其抛出一个异常。

练习 18.10：编写程序令其对两个 ISBN 编号不相同的对象执行 Sales_data 的加法运算。为该程序编写两个不同的版本：一个处理异常，另一个不处理异常。观察并比较这两个程序的行为，用心体会当出现了一个未被捕获的异常时程序会发生什么情况。

练习 18.11：为什么 what 函数不应该抛出异常？

18.2 命名空间

大型程序往往会使用多个独立开发的库，这些库又会定义大量的全局名字，如类、函数和模板等。当应用程序用到多个供应商提供的库时，不可避免地会发生某些名字相互冲突的情况。多个库将名字放置在全局命名空间中将引发**命名空间污染**（namespace pollution）。

传统上，程序员通过将其定义的全局实体名字设得很长来避免命名空间污染问题，这样的名字中通常包含表示名字所属库的前缀部分：

```
class cplusplus_primer_Query { ... };
string cplusplus_primer_make_plural(size_t, string&);
```

这种解决方案显然不太理想：对于程序员来说，书写和阅读这么长的名字费时费力且过于烦琐。

命名空间（namespace）为防止名字冲突提供了更加可控的机制。命名空间分割了全局命名空间，其中每个命名空间是一个作用域。通过在某个命名空间中定义库的名字，库的作者（以及用户）可以避免全局名字固有的限制。

18.2.1 命名空间定义

一个命名空间的定义包含两部分：首先是关键字 namespace，随后是命名空间的名字。在命名空间名字后面是一系列由花括号括起来的声明和定义。只要能出现在全局作用域中的声明就能置于命名空间内，主要包括：类、变量（及其初始化操作）、函数（及其定义）、模板和其他命名空间：

```
namespace cplusplus_primer {
    class Sales_data { /* ...*/};
```

```
    Sales_data operator+(const Sales_data&,
                         const Sales_data&);
    class Query { /* ...*/ };
    class Query_base { /* ...*/};
} // 命名空间结束后无须分号，这一点与块类似
```

786 上面的代码定义了一个名为 cplusplus_primer 的命名空间，该命名空间包含四个成员：三个类和一个重载的+运算符。

和其他名字一样，命名空间的名字也必须在定义它的作用域内保持唯一。命名空间既可以定义在全局作用域内，也可以定义在其他命名空间中，但是不能定义在函数或类的内部。



命名空间作用域后面无须分号。

每个命名空间都是一个作用域

和其他作用域类似，命名空间中的每个名字都必须表示该空间内的唯一实体。因为不同命名空间的作用域不同，所以在不同命名空间内可以有相同名字的成员。

定义在某个命名空间中的名字可以被该命名空间内的其他成员直接访问，也可以被这些成员内嵌作用域中的任何单位访问。位于该命名空间之外的代码则必须明确指出所用的名字属于哪个命名空间：

```
cplusplus_primer::Query q =
                cplusplus_primer::Query("hello");
```

如果其他命名空间（比如说 AddisonWesley）也提供了一个名为 Query 的类，并且我们希望使用这个类替代 cplusplus_primer 中定义的同名类，则可以按照如下方式修改代码：

```
AddisonWesley::Query q = AddisonWesley::Query("hello");
```

命名空间可以是不连续的

如我们在 16.5 节（第 626 页）介绍过的，命名空间可以定义在几个不同的部分，这一点与其他作用域不太一样。编写如下的命名空间定义：

```
namespace nsp {
// 相关声明
}
```

可能是定义了一个名为 nsp 的新命名空间，也可能是为已经存在的命名空间添加一些新成员。如果之前没有名为 nsp 的命名空间定义，则上述代码创建一个新的命名空间；否则，上述代码打开已经存在的命名空间定义并为其添加一些新成员的声明。

命名空间的定义可以不连续的特性使得我们可以将几个独立的接口和实现文件组成一个命名空间。此时，命名空间的组织方式类似于我们管理自定义类及函数的方式：

- 命名空间的一部分成员的作用是定义类，以及声明作为类接口的函数及对象，则这些成员应该置于头文件中，这些头文件将被包含在使用了这些成员的文件中。
- 命名空间成员的定义部分则置于另外的源文件中。

787 在程序中某些实体只能定义一次：如非内联函数、静态数据成员、变量等，命名空间中定义的名字也需要满足这一要求，我们可以通过上面的方式组织命名空间并达到目的。这种接口和实现分离的机制确保我们所需的函数和其他名字只定义一次，而只要是用到这些实

体的地方都能看到对于实体名字的声明。



定义多个类型不相关的命名空间应该使用单独的文件分别表示每个类型（或关联类型构成的集合）。

定义本书的命名空间

通过使用上述接口与实现分离的机制，我们可以将 cplusplus_primer 库定义在几个不同的文件中。Sales_data 类的声明及其函数将置于 Sales_data.h 头文件中，第 15 章介绍的 Query 类将置于 Query.h 头文件中，以此类推。对应的实现文件将分别是 Sales_data.cc 和 Query.cc：

```
// ---- Sales_data.h ----
// #include 应该出现在打开命名空间的操作之前
#include <string>
namespace cplusplus_primer {
    class Sales_data { /* ... */};
    Sales_data operator+(const Sales_data&,
                         const Sales_data&);
    // Sales_data 的其他接口函数的声明
}
// ---- Sales_data.cc ----
// 确保#include 出现在打开命名空间的操作之前
#include "Sales_data.h"

namespace cplusplus_primer {
// Sales_data 成员及重载运算符的定义
}
```

程序如果想使用我们定义的库，必须包含必要的头文件，这些头文件中的名字定义在命名空间 cplusplus_primer 内：

```
// ---- user.cc ----
// Sales_data.h 头文件的名字位于命名空间 cplusplus_primer 中
#include "Sales_data.h"
int main()
{
    using cplusplus_primer::Sales_data;
    Sales_data trans1, trans2;
    // ...
    return 0;
}
```

这种程序的组织方式提供了开发者和库用户所需的模块性。每个类仍组织在自己的接口和实现文件中，一个类的用户不必编译与其他类相关的名字。我们对用户隐藏了实现细节，同时允许文件 Sales_data.cc 和 user.cc 被编译并链接成一个程序而不会产生任何编译时错误或链接时错误。库的开发者可以分别实现每一个类，相互之间没有干扰。

788

有一点需要注意，在通常情况下，我们不把#include 放在命名空间内部。如果我们这么做了，隐含的意思是把头文件中所有的名字定义成该命名空间的成员。例如，如果 Sales_data.h 在包含 string 头文件前就已经打开了命名空间 cplusplus_primer，则程序将出错，因为这么做意味着我们试图将命名空间 std 嵌套在命名空间 cplusplus_primer 中。

定义命名空间成员

假定作用域中存在合适的声明语句，则命名空间中的代码可以使用同一命名空间定义的名字的简写形式：

```
#include "Sales_data.h"
namespace cplusplus_primer {      // 重新打开命名空间 cplusplus_primer
// 命名空间中定义的成员可以直接使用名字，此时无须前缀
std::istream&
operator>>(std::istream& in, Sales_data& s) { /* ...*/}
}
```

也可以在命名空间定义的外部定义该命名空间的成员。命名空间对于名字的声明必须在作用域内，同时该名字的定义需要明确指出其所属的命名空间：

```
// 命名空间之外定义的成员必须使用含有前缀的名字
cplusplus_primer::Sales_data
cplusplus_primer::operator+(const Sales_data& lhs,
                            const Sales_data& rhs)
{
    Sales_data ret(lhs);
    // ...
}
```

和定义在类外部的类成员一样，一旦看到含有完整前缀的名字，我们就可以确定该名字位于命名空间的作用域内。在命名空间 cplusplus_primer 内部，我们可以直接使用该命名空间的其他成员，比如在上面的代码中，可以直接使用 Sales_data 定义函数的形参。

尽管命名空间的成员可以定义在命名空间外部，但是这样的定义必须出现在所属命名空间的外层空间中。换句话说，我们可以在 cplusplus_primer 或全局作用域中定义 Sales_data operator+，但是不能在一个不相关的作用域中定义这个运算符。

模板特例化

模板特例化必须定义在原始模板所属的命名空间中（参见 16.5 节，第 626 页）。和其
789 他命名空间名字类似，只要我们在命名空间中声明了特例化，就能在命名空间外部定义它了：

```
// 我们必须将模板特例化声明成 std 的成员
namespace std {
    template <> struct hash<Sales_data>;
}
// 在 std 中添加了模板特例化的声明后，就可以在命名空间 std 的外部定义它了
template <> struct std::hash<Sales_data>
{
    size_t operator()(const Sales_data& s) const
    { return hash<string>()(s.bookNo) ^
             hash<unsigned>()(s.units_sold) ^
             hash<double>()(s.revenue); }
    // 其他成员与之前的版本一致
};
```

全局命名空间

全局作用域中定义的名字（即在所有类、函数及命名空间之外定义的名字）也就是定义在**全局命名空间**（global namespace）中。全局命名空间以隐式的方式声明，并且在所有

程序中都存在。全局作用域中定义的名字被隐式地添加到全局命名空间中。

作用域运算符同样可以用于全局作用域的成员，因为全局作用域是隐式的，所以它并没有名字。下面的形式

```
::member_name
```

表示全局命名空间中的一个成员。

嵌套的命名空间

嵌套的命名空间是指定义在其他命名空间中的命名空间：

```
namespace cplusplus_primer {
    // 第一个嵌套的命名空间：定义了库的 Query 部分
    namespace QueryLib {
        class Query { /* ... */ };
        Query operator&(const Query&, const Query&);
        // ...
    }
    // 第二个嵌套的命名空间：定义了库的 Sales_data 部分
    namespace Bookstore {
        class Quote { /* ... */ };
        class Disc_quote : public Quote { /* ... */ };
        // ...
    }
}
```

上面的代码将命名空间 cplusplus_primer 分割为两个嵌套的命名空间，分别是 QueryLib 和 Bookstore。

嵌套的命名空间同时是一个嵌套的作用域，它嵌套在外层命名空间的作用域中。嵌套的命名空间中的名字遵循的规则与往常类似：内层命名空间声明的名字将隐藏外层命名空间声明的同名成员。在嵌套的命名空间中定义的名字只在内层命名空间中有效，外层命名空间中的代码要想访问它必须在名字前添加限定符。例如，在嵌套的命名空间 QueryLib 中声明的类名是 790

```
cplusplus_primer::QueryLib::Query
```

内联命名空间

C++11 新标准引入了一种新的嵌套命名空间，称为**内联命名空间**（inline namespace）。和普通的嵌套命名空间不同，内联命名空间中的名字可以被外层命名空间直接使用。也就是说，我们无须在内联命名空间的名字前添加表示该命名空间的前缀，通过外层命名空间的名字就可以直接访问它。 C++11

定义内联命名空间的方式是在关键字 namespace 前添加关键字 inline：

```
inline namespace FifthEd {
    // 该命名空间表示本书第 5 版的代码
}
namespace FifthEd {              // 隐式内联
    class Query_base { /* ... */};
    // 其他与 Query 有关的声明
}
```

关键字 inline 必须出现在命名空间第一次定义的地方，后续再打开命名空间的时候可以写 inline，也可以不写。

当应用程序的代码在一次发布和另一次发布之间发生了改变时，常常会用到内联命名空间。例如，我们可以把本书当前版本的所有代码都放在一个内联命名空间中，而之前版本的代码都放在一个非内联命名空间中：

```
namespace FourthEd {
    class Item_base { /* ... */};
    class Query_base { /* ... */};
    // 本书第 4 版用到的其他代码
}
```

命名空间 cplusplus_primer 将同时使用这两个命名空间。例如，假定每个命名空间都定义在同名的头文件中，则我们可以把命名空间 cplusplus_primer 定义成如下形式：

```
namespace cplusplus_primer {
#include "FifthEd.h"
#include "FourthEd.h"
}
```

791 因为 FifthEd 是内联的，所以形如 cplusplus_primer::的代码可以直接获得 FifthEd 的成员。如果我们想使用早期版本的代码，则必须像其他嵌套的命名空间一样加上完整的外层命名空间名字，比如 cplusplus_primer::FourthEd::Query_base。

未命名的命名空间

未命名的命名空间（unnamed namespace）是指关键字 namespace 后紧跟花括号括起来的一系列声明语句。未命名的命名空间中定义的变量拥有静态生命周期：它们在第一次使用前创建，并且直到程序结束才销毁。

一个未命名的命名空间可以在某个给定的文件内不连续，但是不能跨越多个文件。每个文件定义自己的未命名的命名空间，如果两个文件都含有未命名的命名空间，则这两个空间互相无关。在这两个未命名的命名空间中可以定义相同的名字，并且这些定义表示的是不同实体。如果一个头文件定义了未命名的命名空间，则该命名空间中定义的名字将在每个包含了该头文件的文件中对应不同实体。



和其他命名空间不同，未命名的命名空间仅在特定的文件内部有效，其作用范围不会横跨多个不同的文件。

定义在未命名的命名空间中的名字可以直接使用，毕竟我们找不到什么命名空间的名字来限定它们；同样的，我们也不能对未命名的命名空间的成员使用作用域运算符。

未命名的命名空间中定义的名字的作用域与该命名空间所在的作用域相同。如果未命名的命名空间定义在文件的最外层作用域中，则该命名空间中的名字一定要与全局作用域中的名字有所区别：

```
int i;                          // i 的全局声明
namespace {
    int i;
}
```

```
// 二义性：i 的定义既出现在全局作用域中，又出现在未嵌套的未命名的命名空间中
i = 10;
```

其他情况下，未命名的命名空间中的成员都属于正确的程序实体。和所有命名空间类似，一个未命名的命名空间也能嵌套在其他命名空间当中。此时，未命名的命名空间中的成员可以通过外层命名空间的名字来访问：

```
namespace local {
    namespace {
        int i;
    }
}
// 正确：定义在嵌套的未命名的命名空间中的 i 与全局作用域中的 i 不同
local::i = 42;
```

未命名的命名空间取代文件中的静态声明

792

在标准 C++引入命名空间的概念之前，程序需要将名字声明成 static 的以使得其对于整个文件有效。在文件中进行静态声明的做法是从 C 语言继承而来的。在 C 语言中，声明为 static 的全局实体在其所在的文件外不可见。



在文件中进行静态声明的做法已经被 C++标准取消了，现在的做法是使用未命名的命名空间。

18.2.1 节练习

练习 18.12：将你为之前各章练习编写的程序放置在各自的命名空间中。也就是说，命名空间 chapter15 包含 Query 程序的代码，命名空间 chapter10 包含 TextQuery 的代码；使用这种结构重新编译 Query 代码示例。

练习 18.13：什么时候应该使用未命名的命名空间？

练习 18.14：假设下面的 operator*声明的是嵌套的命名空间 mathLib::MatrixLib 的一个成员：

```
namespace mathLib {
    namespace MatrixLib {
        class matrix { /* ... */ };
        matrix operator*
               (const matrix &, const matrix &);
        // ...
    }
}
```

请问你应该如何在全局作用域中声明该运算符？

18.2.2 使用命名空间成员

像 namespace_name::member_name 这样使用命名空间的成员显然非常烦琐，特别是当命名空间的名字很长时尤其如此。幸运的是，我们可以通过一些其他更简便的方法使用命名空间的成员。之前的程序已经使用过其中一种方法，即 using 声明（参见 3.1 节，第 74 页）。本节还将介绍另外几种方法，如命名空间的别名以及 using 指示等。

命名空间的别名

命名空间的别名（namespace alias）使得我们可以为命名空间的名字设定一个短得多的同义词。例如，一个很长的命名空间的名字形如

```
namespace cplusplus_primer { /* ...*/ };
```

我们可以为其设定一个短得多的同义词：

```
namespace primer = cplusplus_primer;
```

793 命名空间的别名声明以关键字 namespace 开始，后面是别名所用的名字、=符号、命名空间原来的名字以及一个分号。不能在命名空间还没有定义前就声明别名，否则将产生错误。

命名空间的别名也可以指向一个嵌套的命名空间：

```
namespace Qlib = cplusplus_primer::QueryLib;
Qlib::Query q;
```



一个命名空间可以有好几个同义词或别名，所有别名都与命名空间原来的名字等价。

using 声明：扼要概述

一条 **using 声明**（using declaration）语句一次只引入命名空间的一个成员。它使得我们可以清楚地知道程序中所用的到底是哪个名字。

using 声明引入的名字遵守与过去一样的作用域规则：它的有效范围从 using 声明的地方开始，一直到 using 声明所在的作用域结束为止。在此过程中，外层作用域的同名实体将被隐藏。未加限定的名字只能在 using 声明所在的作用域以及其内层作用域中使用。在有效作用域结束后，我们就必须使用完整的经过限定的名字了。

一条 using 声明语句可以出现在全局作用域、局部作用域、命名空间作用域以及类的作用域中。在类的作用域中，这样的声明语句只能指向基类成员（参见 15.5 节，第 546 页）。

using 指示

using 指示（using directive）和 using 声明类似的地方是，我们可以使用命名空间名字的简写形式；和 using 声明不同的地方是，我们无法控制哪些名字是可见的，因为所有名字都是可见的。

using 指示以关键字 using 开始，后面是关键字 namespace 以及命名空间的名字。如果这里所用的名字不是一个已经定义好的命名空间的名字，则程序将发生错误。using 指示可以出现在全局作用域、局部作用域和命名空间作用域中，但是不能出现在类的作用域中。

using 指示使得某个特定的命名空间中所有的名字都可见，这样我们就无须再为它们添加任何前缀限定符了。简写的名字从 using 指示开始，一直到 using 指示所在的作用域结束都能使用。



如果我们提供了一个对 std 等命名空间的 using 指示而未做任何特殊控制的话，将重新引入由于使用了多个库而造成的名字冲突问题。

using 指示与作用域

using 指示引入的名字的作用域远比 using 声明引入的名字的作用域复杂。如我们所知，using 声明的名字的作用域与 using 声明语句本身的作用域一致，从效果上看就好像 using 声明语句为命名空间的成员在当前作用域内创建了一个别名一样。

using 指示所做的绝非声明别名这么简单。相反，它具有将命名空间成员提升到包含命名空间本身和 using 指示的最近作用域的能力。 794

using 声明和 using 指示在作用域上的区别直接决定了它们工作方式的不同。对于 using 声明来说，我们只是简单地令名字在局部作用域内有效。相反，using 指示是令整个命名空间的所有内容变得有效。通常情况下，命名空间中会含有一些不能出现在局部作用域中的定义，因此，using 指示一般被看作是出现在最近的外层作用域中。

在最简单的情况下，假定我们有一个命名空间 A 和一个函数 f，它们都定义在全局作用域中。如果 f 含有一个对 A 的 using 指示，则在 f 看来，A 中的名字仿佛是出现在全局作用域中 f 之前的位置一样：

```
// 命名空间 A 和函数 f 定义在全局作用域中
namespace A {
    int i, j;
}
void f()
{
    using namespace A;              // 把 A 中的名字注入到全局作用域中
    cout << i * j << endl;          // 使用命名空间 A 中的 i 和 j
    // ...
}
```

using 指示示例

让我们看一个简单的示例：

```
namespace blip {
    int i = 16, j = 15, k = 23;
    // 其他声明
}
int j = 0;                  // 正确：blip 的 j 隐藏在命名空间中
void manip()
{
    // using 指示，blip 中的名字被“添加”到全局作用域中
    using namespace blip; // 如果使用了 j，则将在::j 和 blip::j 之间产生冲突
    ++i;                        // 将 blip::i 设定为 17
    ++j;                        // 二义性错误：是全局的 j 还是 blip::j?
    ++::j;                      // 正确：将全局的 j 设定为 1
    ++blip::j;                  // 正确：将 blip::j 设定为 16
    int k = 97;                 // 当前局部的 k 隐藏了 blip::k
    ++k;                        // 将当前局部的 k 设定为 98
}
```

manip 的 using 指示使得程序可以直接访问 blip 的所有名字，也就是说，manip 的代码可以使用 blip 中名字的简写形式。

blip 的成员看起来好像是定义在 blip 和 manip 所在的作用域一样。假定 manip 795

定义在全局作用域中，则 blip 的成员也好像是定义在全局作用域中一样。

当命名空间被注入到它的外层作用域之后，很有可能该命名空间中定义的名字会与其外层作用域中的成员冲突。例如在 manip 中，blip 的成员 j 就与全局作用域中的 j 产生了冲突。这种冲突是允许存在的，但是要想使用冲突的名字，我们就必须明确指出名字的版本。manip 中所有未加限定的 j 都会产生二义性错误。

为了使用像 j 这样的名字，我们必须使用作用域运算符来明确指出所需的版本。我们使用 ::j 来表示定义在全局作用域中的 j，而使用 blip::j 来表示定义在 blip 中的 j。

因为 manip 的作用域和命名空间的作用域不同，所以 manip 内部的声明可以隐藏命名空间中的某些成员名字。例如，局部变量 k 隐藏了命名空间的成员 blip::k。在 manip 内使用 k 不存在二义性，它指的就是局部变量 k。

头文件与 using 声明或指示

头文件如果在其顶层作用域中含有 using 指示或 using 声明，则会将名字注入到所有包含了该头文件的文件中。通常情况下，头文件应该只负责定义接口部分的名字，而不定义实现部分的名字。因此，头文件最多只能在它的函数或命名空间内使用 using 指示或 using 声明（参见 3.1 节，第 75 页）。

提示：避免 using 指示

using 指示一次性注入某个命名空间的所有名字，这种用法看似简单实则充满了风险：只使用一条语句就突然将命名空间中所有成员的名字变得可见了。如果应用程序使用了多个不同的库，而这些库中的名字通过 using 指示变得可见，则全局命名空间污染的问题将重新出现。

而且，当引入库的新版本后，正在工作的程序很可能会编译失败。如果新版本引入了一个与应用程序正在使用的名字冲突的名字，就会出现这个问题。

另一个风险是由 using 指示引发的二义性错误只有在使用了冲突名字的地方才能被发现。这种延后的检测意味着可能在特定库引入很久之后才爆发冲突。直到程序开始使用该库的新部分后，之前一直未被检测到的错误才会出现。

相比于使用 using 指示，在程序中对命名空间的每个成员分别使用 using 声明效果更好，这么做可以减少注入到命名空间中的名字数量。using 声明引起的二义性问题在声明处就能发现，无须等到使用名字的地方，这显然对检测并修改错误大有益处。



using 指示也并非一无是处，例如在命名空间本身的实现文件中就可以使用 using 指示。

796

18.2.2 节练习

练习 18.15：说明 using 指示与 using 声明的区别。

练习 18.16：假定在下面的代码中标记为“位置 1”的地方是对于命名空间 Exercise 中所有成员的 using 声明，请解释代码的含义。如果这些 using 声明出现在“位置 2”又会怎样呢？将 using 声明变为 using 指示，重新回答之前的问题。

```
namespace Exercise {
    int ivar = 0;
    double dvar = 0;
```

```
        const int limit = 1000;
    }
    int ivar = 0;
    // 位置 1
    void manip() {
        // 位置 2
        double dvar = 3.1416;
        int iobj = limit + 1;
        ++ivar;
        ++::ivar;
    }
```

练习 18.17：实际编写代码检验你对上一题的回答是否正确。

18.2.3 类、命名空间与作用域

对命名空间内部名字的查找遵循常规的查找规则：即由内向外依次查找每个外层作用域。外层作用域也可能是一个或多个嵌套的命名空间，直到最外层的全局命名空间查找过程终止。只有位于开放的块中且在使用点之前声明的名字才被考虑：

```
namespace A {
    int i;
    namespace B {
        int i;                  // 在 B 中隐藏了 A::i
        int j;
        int f1()
        {
            int j;              // j 是 f1 的局部变量，隐藏了 A::B::j
            return i;           // 返回 B::i
        }
    } // 命名空间 B 结束，此后 B 中定义的名字不再可见
    int f2() {
       return j;                // 错误：j 没有被定义
    }
    int j = i;                  // 用 A::i 进行初始化
}
```

对于位于命名空间中的类来说，常规的查找规则仍然适用：当成员函数使用某个名字时，首先在该成员中进行查找，然后在类中查找（包括基类），接着在外层作用域中查找，这时一个或几个外层作用域可能就是命名空间：

797

```
namespace A {
    int i;
    int k;
    class C1 {
    public:
        C1(): i(0), j(0) { }        // 正确：初始化 C1::i 和 C1::j
        int f1() { return k; }      // 返回 A::k
        int f2() { return h; }      // 错误：h 未定义
        int f3();
    private:
        int i;                      // 在 C1 中隐藏了 A::i
```

```
        int j;
    };
    int h = i;                          // 用 A::i 进行初始化
}
// 成员 f3 定义在 C1 和命名空间 A 的外部
int A::C1::f3() { return h; }           // 正确：返回 A::h
```

除了类内部出现的成员函数定义之外（参见 7.4.1 节，第 254 页），总是向上查找作用域。名字必须先声明后使用，因此 f2 的 return 语句无法通过编译。该语句试图使用命名空间 A 的名字 h，但此时 h 尚未定义。如果 h 在 A 中定义的位置位于 C1 的定义之前，则上述语句将合法。类似的，因为 f3 的定义位于 A::h 之后，所以 f3 对于 h 的使用是合法的。



可以从函数的限定名推断出查找名字时检查作用域的次序，限定名以相反次序指出被查找的作用域。

限定符 A::C1::f3 指出了查找类作用域和命名空间作用域的相反次序。首先查找函数 f3 的作用域，然后查找外层类 C1 的作用域，最后检查命名空间 A 的作用域以及包含着 f3 定义的作用域。

实参相关的查找与类类型形参

考虑下面这个简单的程序：

```
std::string s;
std::cin >> s;
```

如我们所知，该调用等价于（参见 14.1 节，第 491 页）：

```
operator>>(std::cin, s);
```

798 operator>>函数定义在标准库 string 中，string 又定义在命名空间 std 中。但是我们不用 std::限定符和 using 声明就可以调用 operator>>。

对于命名空间中名字的隐藏规则来说有一个重要的例外，它使得我们可以直接访问输出运算符。这个例外是，当我们给函数传递一个类类型的对象时，除了在常规的作用域查找外还会查找实参类所属的命名空间。这一例外对于传递类的引用或指针的调用同样有效。

在此例中，当编译器发现对 operator>>的调用时，首先在当前作用域中寻找合适的函数，接着查找输出语句的外层作用域。随后，因为>>表达式的形参是类类型的，所以编译器还会查找 cin 和 s 的类所属的命名空间。也就是说，对于这个调用来说，编译器会查找定义了 istream 和 string 的命名空间 std。当在 std 中查找时，编译器找到了 string 的输出运算符函数。

查找规则的这个例外允许概念上作为类接口一部分的非成员函数无须单独的 using 声明就能被程序使用。假如该例外不存在，则我们将不得不为输出运算符专门提供一个 using 声明：

```
using std::operator>>;              // 要想使用 cin >> s 就必须有该 using 声明
```

或者使用函数调用的形式以把命名空间的信息包含进来：

```
std::operator>>(std::cin, s);       // 正确：显式地使用 std::>>
```

在没有使用运算符语法的情况下，上述两种声明都显得比较笨拙且无形中增加了使用 IO 标准库的难度。

查找与 std::move 和 std::forward

很多甚至是绝大多数 C++程序员从来都没有考虑过与实参相关的查找问题。通常情况下，如果在应用程序中定义了一个标准库中已有的名字，则将出现以下两种情况中的一种：要么根据一般的重载规则确定某次调用应该执行函数的哪个版本；要么应用程序根本就不会执行函数的标准库版本。

接下来考虑标准库 `move` 和 `forward` 函数。这两个都是模板函数，在标准库的定义中它们都接受一个右值引用的函数形参。如我们所知，在函数模板中，右值引用形参可以匹配任何类型（参见 16.2.6 节，第 611 页）。如果我们的应用程序也定义了一个接受单一形参的 `move` 函数，则不管该形参是什么类型，应用程序的 `move` 函数都将与标准库的版本冲突。`forward` 函数也是如此。

因此，`move`（以及 `forward`）的名字冲突要比其他标准库函数的冲突频繁得多。而且，因为 `move` 和 `forward` 执行的是非常特殊的类型操作，所以应用程序专门修改函数原有行为的概率非常小。

对于 `move` 和 `forward` 来说，冲突很多但是大多数是无意的，这一特点解释了为什么我们建议最好使用它们的带限定语的完整版本的原因（参见 12.1.5 节，第 417 页）。通过书写 `std::move` 而非 `move`，我们就能明确地知道想要使用的是函数的标准库版本。 799

友元声明与实参相关的查找



回顾我们曾经讨论过的，当类声明了一个友元时，该友元声明并没有使得友元本身可见（参见 7.2.1 节，第 242 页）。然而，一个另外的未声明的类或函数如果第一次出现在友元声明中，则我们认为它是最近的外层命名空间的成员。这条规则与实参相关的查找规则结合在一起将产生意想不到的效果：

```
namespace A {
    class C {
        // 两个友元，在友元声明之外没有其他的声明
        // 这些函数隐式地成为命名空间 A 的成员
        friend void f2();          // 除非另有声明，否则不会被找到
        friend void f(const C&); // 根据实参相关的查找规则可以被找到
    };
}
```

此时，`f` 和 `f2` 都是命名空间 `A` 的成员。即使 `f` 不存在其他声明，我们也能通过实参相关的查找规则调用 `f`：

```
int main()
{
    A::C cobj;
    f(cobj);                    // 正确：通过在 A::C 中的友元声明找到 A::f
    f2();                       // 错误：A::f2 没有被声明
}
```

因为 `f` 接受一个类类型的实参，而且 `f` 在 `C` 所属的命名空间进行了隐式的声明，所以 `f` 能被找到。相反，因为 `f2` 没有形参，所以它无法被找到。

18.2.3 节练习

练习 18.18：已知有下面的 swap 的典型定义（参见 13.3 节，第 457 页），当 mem1 是一个 string 时程序使用 swap 的哪个版本？如果 mem1 是 int 呢？说明在这两种情况下名字查找的过程。

```
void swap(T v1, T v2)
{
    using std::swap;
    swap(v1.mem1, v2.mem1);
    // 交换类型 T 的其他成员
}
```

练习 18.19：如果对 swap 的调用形如 std::swap(v1.mem1, v2.mem1)将发生什么情况？

800

18.2.4 重载与命名空间

命名空间对函数的匹配过程有两方面的影响（参见 6.4 节，第 209 页）。其中一个影响非常明显：using 声明或 using 指示能将某些函数添加到候选函数集。另外一个影响则比较微妙。

与实参相关的查找与重载

在上一节中我们了解到，对于接受类类型实参的函数来说，其名字查找将在实参类所属的命名空间中进行。这条规则对于我们如何确定候选函数集同样也有影响。我们将在每个实参类（以及实参类的基类）所属的命名空间中搜寻候选函数。在这些命名空间中所有与被调用函数同名的函数都将被添加到候选集当中，即使其中某些函数在调用语句处不可见也是如此：

```
namespace NS {
    class Quote { /* ... */ };
    void display(const Quote&) { /* ... */ }
}
// Bulk_item 的基类声明在命名空间 NS 中
class Bulk_item : public NS::Quote { /* ... */ };
int main() {
    Bulk_item book1;
    display(book1);
    return 0;
}
```

我们传递给 display 的实参属于类类型 Bulk_item，因此该调用语句的候选函数不仅应该在调用语句所在的作用域中查找，而且也应该在 Bulk_item 及其基类 Quote 所属的命名空间中查找。命名空间 NS 中声明的函数 display(const Quote&)也将被添加到候选函数集当中。

重载与 using 声明

要想理解 using 声明与重载之间的交互关系，必须首先明确一条：using 声明语句声明的是一个名字，而非一个特定的函数（参见 15.6 节，第 551 页）：

```
using NS::print(int);     // 错误：不能指定形参列表
using NS::print;          // 正确：using 声明只声明一个名字
```

当我们为函数书写 using 声明时，该函数的所有版本都被引入到当前作用域中。

一个 using 声明囊括了重载函数的所有版本以确保不违反命名空间的接口。库的作者为某项任务提供了好几个不同的函数，允许用户选择性地忽略重载函数中的一部分但不是全部有可能导致意想不到的程序行为。

一个 using 声明引入的函数将重载该声明语句所属作用域中已有的其他同名函数。 801
如果 using 声明出现在局部作用域中，则引入的名字将隐藏外层作用域的相关声明。如果 using 声明所在的作用域中已经有一个函数与新引入的函数同名且形参列表相同，则该 using 声明将引发错误。除此之外，using 声明将为引入的名字添加额外的重载实例，并最终扩充候选函数集的规模。

重载与 using 指示

using 指示将命名空间的成员提升到外层作用域中，如果命名空间的某个函数与该命名空间所属作用域的函数同名，则命名空间的函数将被添加到重载集合中：

```
namespace libs_R_us {
    extern void print(int);
    extern void print(double);
}
// 普通的声明
void print(const std::string &);
// 这个 using 指示把名字添加到 print 调用的候选函数集
using namespace libs_R_us;
// print 调用此时的候选函数包括：
// libs_R_us 的 print(int)
// libs_R_us 的 print(double)
// 显式声明的 print(const std::string &)
void fooBar(int ival)
{
    print("Value: ");          // 调用全局函数 print(const string &)
    print(ival);               // 调用 libs_R_us::print(int)
}
```

与 using 声明不同的是，对于 using 指示来说，引入一个与已有函数形参列表完全相同的函数并不会产生错误。此时，只要我们指明调用的是命名空间中的函数版本还是当前作用域的版本即可。

跨越多个 using 指示的重载

如果存在多个 using 指示，则来自每个命名空间的名字都会成为候选函数集的一部分：

```
namespace AW {
    int print(int);
}
namespace Primer {
    double print(double);
}
// using 指示从不同的命名空间中创建了一个重载函数集合
using namespace AW;
using namespace Primer;
long double print(long double);
int main() {
```

802

```
    print(1);              // 调用 AW::print(int)
    print(3.1);            // 调用 Primer::print(double)
    return 0;
}
```

在全局作用域中，函数 print 的重载集合包括 print(int)、print(double)和 print(long double)，尽管它们的声明位于不同作用域中，但它们都属于 main 函数中 print 调用的候选函数集。

18.2.4 节练习

练习 18.20：在下面的代码中，确定哪个函数与 compute 调用匹配。列出所有候选函数和可行函数，对于每个可行函数的实参与形参的匹配过程来说，发生了哪种类型转换？

```
namespace primerLib {
    void compute();
    void compute(const void *);
}
using primerLib::compute;
void compute(int);
void compute(double, double = 3.4);
void compute(char*, char* = 0);
void f()
{
    compute(0);
}
```

如果将 using 声明置于 main 函数中 compute 的调用点之前将发生什么情况？重新回答之前的那些问题。

18.3 多重继承与虚继承

多重继承（multiple inheritance）是指从多个直接基类（参见 15.2.2 节，第 533 页）中产生派生类的能力。多重继承的派生类继承了所有父类的属性。尽管概念上非常简单，但是多个基类相互交织产生的细节可能会带来错综复杂的设计问题与实现问题。

为了探讨有关多重继承的问题，我们将以动物园中动物的层次关系作为教学实例。动物园中的动物存在于不同的抽象级别上。有个体的动物，如 Ling-Ling、Mowgli 和 Balou 等，它们以名字进行区分；每个动物属于一个物种，例如 Ling-Ling 是一只大熊猫；物种又是科的成员，大熊猫是熊科的成员；每个科是动物界的成员，在这个例子中动物界是指一个动物园中所有动物的总和。

803

我们将定义一个抽象类 ZooAnimal，用它来保存动物园中动物共有的信息并提供公共接口。类 Bear 将存放 Bear 科特有的信息，以此类推。

除了类 ZooAnimal 之外，我们的应用程序还包含其他一些辅助类，这些类负责封装不同的抽象，如濒临灭绝的动物。以类 Panda 的实现为例，Panda 是由 Bear 和 Endangered 共同派生而来的。

18.3.1 多重继承

在派生类的派生列表中可以包含多个基类：

```
class Bear : public ZooAnimal {
class Panda : public Bear, public Endangered { /* ...*/ };
```

每个基类包含一个可选的访问说明符（参见 15.5 节，第 543 页）。一如往常，如果访问说明符被忽略掉了，则关键字 class 对应的默认访问说明符是 private，关键字 struct 对应的是 public（参见 15.5 节，第 546 页）。

和只有一个基类的继承一样，多重继承的派生列表也只能包含已经被定义过的类，而且这些类不能是 final 的（参见 15.2.2 节，第 533 页）。对于派生类能够继承的基类个数，C++没有进行特殊规定；但是在某个给定的派生列表中，同一个基类只能出现一次。

多重继承的派生类从每个基类中继承状态

在多重继承关系中，派生类的对象包含有每个基类的子对象（参见 15.2.2 节，第 530 页）。如图 18.2 所示，在 Panda 对象中含有一个 Bear 部分（其中又含有一个 ZooAnimal 部分）、一个 Endangered 部分以及在 Panda 中声明的非静态数据成员。

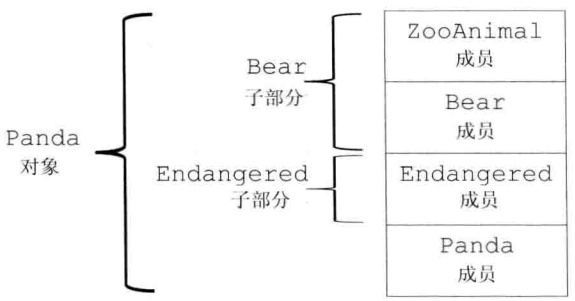


图 18.2：Panda 对象的概念结构

派生类构造函数初始化所有基类

804

构造一个派生类的对象将同时构造并初始化它的所有基类子对象。与从一个基类进行的派生一样（参见 15.2.2 节，第 531 页），多重继承的派生类的构造函数初始值也只能初始化它的直接基类：

```
// 显式地初始化所有基类
Panda::Panda(std::string name, bool onExhibit)
      : Bear(name, onExhibit, "Panda"),
        Endangered(Endangered::critical) { }
// 隐式地使用 Bear 的默认构造函数初始化 Bear 子对象
Panda::Panda()
      : Endangered(Endangered::critical) { }
```

派生类的构造函数初始值列表将实参分别传递给每个直接基类。其中基类的构造顺序与派生列表中基类的出现顺序保持一致，而与派生类构造函数初始值列表中基类的顺序无关。一个 Panda 对象按照如下次序进行初始化：

- ZooAnimal 是整个继承体系的最终基类，Bear 是 Panda 的直接基类，ZooAnimal 是 Bear 的基类，所以首先初始化 ZooAnimal。
- 接下来初始化 Panda 的第一个直接基类 Bear。

- 然后初始化 Panda 的第二个直接基类 Endangered。
- 最后初始化 Panda。

继承的构造函数与多重继承

C++11 在 C++11 新标准中，允许派生类从它的一个或几个基类中继承构造函数（参见 15.7.4 节，第 557 页）。但是如果从多个基类中继承了相同的构造函数（即形参列表完全相同），则程序将产生错误：

```
struct Base1 {
    Base1() = default;
    Base1(const std::string&);
    Base1(std::shared_ptr<int>);
};
struct Base2 {
    Base2() = default;
    Base2(const std::string&);
    Base2(int);
};
// 错误：D1 试图从两个基类中都继承 D1::D1(const string&)
struct D1: public Base1, public Base2 {
    using Base1::Base1;          // 从 Base1 继承构造函数
    using Base2::Base2;          // 从 Base2 继承构造函数
};
```

如果一个类从它的多个基类中继承了相同的构造函数，则这个类必须为该构造函数定义它自己的版本：

805

```
struct D2: public Base1, public Base2 {
    using Base1::Base1;          // 从 Base1 继承构造函数
    using Base2::Base2;          // 从 Base2 继承构造函数
    // D2 必须自定义一个接受 string 的构造函数
    D2(const string &s): Base1(s), Base2(s) { }
    D2() = default;              // 一旦 D2 定义了它自己的构造函数，则必须出现
};
```

析构函数与多重继承

和往常一样，派生类的析构函数只负责清除派生类本身分配的资源，派生类的成员及基类都是自动销毁的。合成的析构函数体为空。

析构函数的调用顺序正好与构造函数相反，在我们的例子中，析构函数的调用顺序是 ~Panda、~Endangered、~Bear 和~ZooAnimal。

多重继承的派生类的拷贝与移动操作

与只有一个基类的继承一样，多重继承的派生类如果定义了自己的拷贝/赋值构造函数和赋值运算符，则必须在完整的对象上执行拷贝、移动或赋值操作（参见 15.7.2 节，第 553 页）。只有当派生类使用的是合成版本的拷贝、移动或赋值成员时，才会自动对其基类部分执行这些操作。在合成的拷贝控制成员中，每个基类分别使用自己的对应成员隐式地完成构造、赋值或销毁等工作。

例如，假设 Panda 使用了合成版本的成员 ling_ling 的初始化过程：

```
Panda ying_yang("ying_yang");
```

```
Panda ling_ling = ying_yang;          // 使用拷贝构造函数
```

将调用 Bear 的拷贝构造函数，后者又在执行自己的拷贝任务之前先调用 ZooAnimal 的拷贝构造函数。一旦 ling_ling 的 Bear 部分构造完成，接着就会调用 Endangered 的拷贝构造函数来创建对象相应的部分。最后，执行 Panda 的拷贝构造函数。合成的移动构造函数的工作机理与之类似。

合成的拷贝赋值运算符的行为与拷贝构造函数很相似。它首先赋值 Bear 部分（并且通过 Bear 赋值 ZooAnimal 部分），然后赋值 Endangered 部分，最后是 Panda 部分。移动赋值运算符的工作机理与之类似。

18.3.1 节练习

练习 18.21：解释下列声明的含义，在它们当中存在错误吗？如果有，请指出来并说明错误的原因。

```
(a) class CADVehicle : public CAD, Vehicle { ... };
(b) class DblList: public List, public List { ... };
(c) class iostream: public istream, public ostream { ... };
```

练习 18.22：已知存在如下所示的类的继承体系，其中每个类都定义了一个默认构造函数：

```
class A { ... };
class B : public A { ... };
class C : public B { ... };
class X { ... };
class Y { ... };
class Z : public X, public Y { ... };
class MI : public C, public Z { ... };
```

对于下面的定义来说，构造函数的执行顺序是怎样的？

```
MI mi;
```

18.3.2 类型转换与多个基类

在只有一个基类的情况下，派生类的指针或引用能自动转换成一个可访问基类的指针或引用（参见 15.2.2 节，第 530 页；参见 15.5 节，第 544 页）。多个基类的情况与之类似。我们可以令某个可访问基类的指针或引用直接指向一个派生类对象。例如，一个 ZooAnimal、Bear 或 Endangered 类型的指针或引用可以绑定到 Panda 对象上：

806

```
// 接受 Panda 的基类引用的一系列操作
void print(const Bear&);
void highlight(const Endangered&);
ostream& operator<<(ostream&, const ZooAnimal&);
Panda ying_yang("ying_yang");
print(ying_yang);           // 把一个 Panda 对象传递给一个 Bear 的引用
highlight(ying_yang);       // 把一个 Panda 对象传递给一个 Endangered 的引用
cout << ying_yang << endl; // 把一个 Panda 对象传递给一个 ZooAnimal 的引用
```

编译器不会在派生类向基类的几种转换中进行比较和选择，因为在它看来转换到任意一种基类都一样好。例如，如果存在如下所示的 print 重载形式：

```
void print(const Bear&);
void print(const Endangered&);
```

则通过 Panda 对象对不带前缀限定符的 print 函数进行调用将产生编译错误：

```
Panda ying_yang("ying_yang");
print(ying_yang);                    // 二义性错误
```

基于指针类型或引用类型的查找

与只有一个基类的继承一样，对象、指针和引用的静态类型决定了我们能够使用哪些
807 成员（参见 15.6 节，第 547 页）。如果我们使用一个 ZooAnimal 指针，则只有定义在 ZooAnimal 中的操作是可以使用的，Panda 接口中的 Bear、Panda 和 Endangered 特有的部分都不可见。类似的，一个 Bear 类型的指针或引用只能访问 Bear 及 ZooAnimal 的成员，一个 Endangered 的指针或引用只能访问 Endangered 的成员。

举个例子，已知我们的类已经定义了表 18.1 列出的虚函数，考虑下面的这些函数调用：

```
Bear *pb = new Panda("ying_yang");
pb->print();                    // 正确：Panda::print()
pb->cuddle();                   // 错误：不属于 Bear 的接口
pb->highlight();                // 错误：不属于 Bear 的接口
delete pb;                      // 正确：Panda::~Panda()
```

当我们通过 Endangered 的指针或引用访问一个 Panda 对象时，Panda 接口中 Panda 特有的部分以及属于 Bear 的部分都是不可见的：

```
Endangered *pe = new Panda("ying_yang");
pe->print();                // 正确：Panda::print()
pe->toes();                 // 错误：不属于 Endangered 的接口
pe->cuddle();               // 错误：不属于 Endangered 的接口
pe->highlight();            // 正确：Panda::highlight()
delete pe;                  // 正确：Panda::~Panda()
```

表 18.1：在 ZooAnimal/Endangered 中定义的虚函数

函数	含有自定义版本的类
print	ZooAnimal::ZooAnimal
	Bear::Bear
	Endangered::Endangered
	Panda::Panda
highlight	Endangered::Endangered
	Panda::Panda
toes	Bear::Bear
	Panda::Panda
cuddle	Panda::Panda
析构函数	ZooAnimal::ZooAnimal
	Endangered::Endangered

18.3.2 节练习

练习 18.23：使用练习 18.22 的继承体系以及下面定义的类 D，同时假定每个类都定义了默认构造函数，请问下面的哪些类型转换是不被允许的？

```
class D : public X, public C { ... };
D *pd = new D;
(a) X *px = pd;        (b) A *pa = pd;
(c) B *pb = pd;        (d) C *pc = pd;
```

练习 18.24：在第 714 页，我们使用一个指向 Panda 对象的 Bear 指针进行了一系列调用，假设我们使用的是一个指向 Panda 对象的 ZooAnimal 指针将发生什么情况，请对这些调用语句逐一进行说明。

练习 18.25：假设我们有两个基类 Base1 和 Base2，它们各自定义了一个名为 print 的虚成员和一个虚析构函数。从这两个基类中我们派生出下面的类，它们都重新定义了 print 函数：

```
class D1 : public Base1 { /* ...*/ };
class D2 : public Base2 { /* ...*/ };
class MI : public D1, public D2 { /* ...*/ };
```

通过下面的指针，指出在每个调用中分别使用了哪个函数：

```
Base1 *pb1 = new MI;
Base2 *pb2 = new MI;
D1 *pd1 = new MI;
D2 *pd2 = new MI;
(a) pb1->print();      (b) pd1->print();      (c) pd2->print();
(d) delete pb2;        (e) delete pd1;        (f) delete pd2;
```

18.3.3 多重继承下的类作用域

在只有一个基类的情况下，派生类的作用域嵌套在直接基类和间接基类的作用域中（参见 15.6 节，第 547 页）。查找过程沿着继承体系自底向上进行，直到找到所需的名字。派生类的名字将隐藏基类的同名成员。

在多重继承的情况下，相同的查找过程在所有直接基类中同时进行。如果名字在多个基类中都被找到，则对该名字的使用将具有二义性。

在我们的例子中，如果我们通过 Panda 的对象、指针或引用使用了某个名字，则程序会并行地在 Endangered 和 Bear/ZooAnimal 这两棵子树中查找该名字。如果名字在超过一棵子树中被找到，则该名字的使用具有二义性。对于一个派生类来说，从它的几个基类中分别继承名字相同的成员是完全合法的，只不过在使用这个名字时必须明确指出它的版本。 808



当一个类拥有多个基类时，有可能出现派生类从两个或更多基类中继承了同名成员的情况。此时，不加前缀限定符直接使用该名字将引发二义性。

例如，如果 ZooAnimal 和 Endangered 都定义了名为 max_weight 的成员，并且 Panda 没有定义该成员，则下面的调用是错误的：

```
double d = ying_yang.max_weight();
```

Panda 在派生的过程中拥有了两个名为 max_weight 的成员，这是完全合法的。派生仅仅是产生了潜在的二义性，只要 Panda 对象不调用 max_weight 函数就能避免二义性错误。另外，如果每次调用 max_weight 时都指出所调用的版本 809

(ZooAnimal::max_weight 或者 Endangered::max_weight)，也不会发生二义性。只有当要调用哪个函数含糊不清时程序才会出错。

在上面的例子中，派生类继承的两个 max_weight 会产生二义性，这一点显而易见。一种更复杂的情况是，有时即使派生类继承的两个函数形参列表不同也可能发生错误。此外，即使 max_weight 在一个类中是私有的，而在另一个类中是公有的或受保护的同样也可能发生错误。最后一种情况，假如 max_weight 定义在 Bear 中而非 ZooAnimal 中，上面的程序仍然是错误的。

和往常一样，先查找名字后进行类型检查（参见 6.4.1 节，第 210 页）。当编译器在两个作用域中同时发现了 max_weight 时，将直接报告一个调用二义性的错误。

要想避免潜在的二义性，最好的办法是在派生类中为该函数定义一个新版本。例如，我们可以为 Panda 定义一个 max_weight 函数从而解决二义性问题：

```
double Panda::max_weight() const
{
    return std::max(ZooAnimal::max_weight(),
                    Endangered::max_weight());
}
```

18.3.3 节练习

```
struct Base1 {
    void print(int) const;           // 默认情况下是公有的
protected:
    int ival;
    double dval;
    char cval;
private:
    int *id;
};
struct Base2 {
    void print(double) const;        // 默认情况下是公有的
protected:
    double fval;
private:
    double dval;
};
struct Derived : public Base1 {
    void print(std::string) const;   // 默认情况下是公有的
protected:
    std::string sval;
    double dval;
};
struct MI : public Derived, public Base2 {
    void print(std::vector<double>); // 默认情况下是公有的
protected:
    int *ival;
    std::vector<double> dvec;
};
```

练习 18.26：已知如上所示的继承体系，下面对 print 的调用为什么是错误的？适当修改 MI，令其对 print 的调用可以编译通过并正确执行。

```
MI mi;
mi.print(42);
```

练习 18.27：已知如上所示的继承体系，同时假定为 MI 添加了一个名为 foo 的函数：

```
int ival;
double dval;
void MI::foo(double cval)
{
    int dval;
    // 练习中的问题发生在此处
}
```

(a) 列出在 MI::foo 中可见的所有名字。
(b) 是否存在某个可见的名字是继承自多个基类的？
(c) 将 Base1 的 dval 成员与 Derived 的 dval 成员求和后赋给 dval 的局部实例。
(d) 将 MI::dvec 的最后一个元素的值赋给 Base2::fval。
(e) 将从 Base1 继承的 cval 赋给从 Derived 继承的 sval 的第一个字符。

18.3.4 虚继承

810

尽管在派生列表中同一个基类只能出现一次，但实际上派生类可以多次继承同一个类。派生类可以通过它的两个直接基类分别继承同一个间接基类，也可以直接继承某个基类，然后通过另一个基类再一次间接继承该类。

举个例子，IO 标准库的 istream 和 ostream 分别继承了一个共同的名为 base_ios 的抽象基类。该抽象基类负责保存流的缓冲内容并管理流的条件状态。iostream 是另外一个类，它从 istream 和 ostream 直接继承而来，可以同时读写流的内容。因为 istream 和 ostream 都继承自 base_ios，所以 iostream 继承了 base_ios 两次，一次是通过 istream，另一次是通过 ostream。

在默认情况下，派生类中含有继承链上每个类对应的子部分。如果某个类在派生过程中出现了多次，则派生类中将包含该类的多个子对象。

这种默认的情况对某些形如 iostream 的类显然是行不通的。一个 iostream 对象肯定希望在同一个缓冲区中进行读写操作，也会要求条件状态能同时反映输入和输出操作的情况。假如在 iostream 对象中真的包含了 base_ios 的两份拷贝，则上述的共享行为就无法实现了。

811

在 C++语言中我们通过**虚继承**（virtual inheritance）的机制解决上述问题。虚继承的目的是令某个类做出声明，承诺愿意共享它的基类。其中，共享的基类子对象称为**虚基类**（virtual base class）。在这种机制下，不论虚基类在继承体系中出现了多少次，在派生类中都只包含唯一一个共享的虚基类子对象。

另一个 Panda 类

在过去，科学界对于大熊猫属于 Raccoon 科还是 Bear 科争论不休。为了如实地反映这种争论，我们可以对 Panda 类进行修改，令其同时继承 Bear 和 Raccoon。此时，为了避免赋予 Panda 两份 ZooAnimal 的子对象，我们将 Bear 和 Raccoon 继承

ZooAnimal 的方式定义为虚继承。图 18.3 描述了新的继承体系。

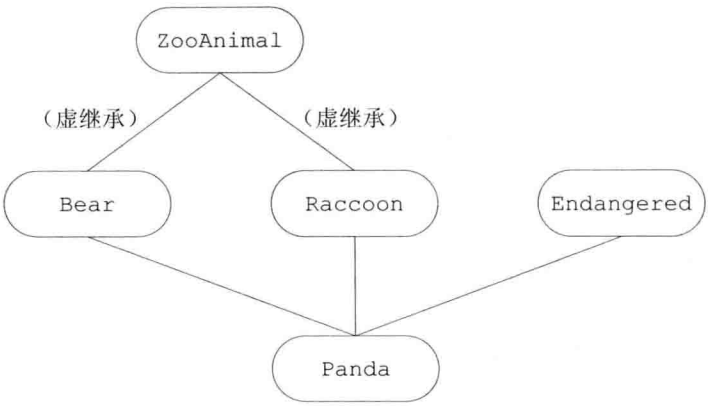


图 18.3：Panda 的虚继承层次

观察这个新的继承体系，我们将发现虚继承的一个不太直观的特征：必须在虚派生的真实需求出现前就已经完成虚派生的操作。例如在我们的类中，当我们定义 Panda 时才出现了对虚派生的需求，但是如果 Bear 和 Raccoon 不是从 ZooAnimal 虚派生得到的，那么 Panda 的设计者就显得不太幸运了。

在实际的编程过程中，位于中间层次的基类将其继承声明为虚继承一般不会带来什么问题。通常情况下，使用虚继承的类层次是由一个人或一个项目组一次性设计完成的。对于一个独立开发的类来说，很少需要基类中的某一个是虚基类，况且新基类的开发者也无法改变已存在的类体系。



虚派生只影响从指定了虚基类的派生类中进一步派生出的类，它不会影响派生类本身。

812

使用虚基类

我们指定虚基类的方式是在派生列表中添加关键字 virtual：

```
// 关键字 public 和 virtual 的顺序随意
class Raccoon : public virtual ZooAnimal { /* ...*/ };
class Bear : virtual public ZooAnimal { /* ...*/ };
```

通过上面的代码我们将 ZooAnimal 定义为 Raccoon 和 Bear 的虚基类。

virtual 说明符表明了一种愿望，即在后续的派生类当中共享虚基类的同一份实例。至于什么样的类能够作为虚基类并没有特殊规定。

如果某个类指定了虚基类，则该类的派生仍按常规方式进行：

```
class Panda : public Bear,
              public Raccoon, public Endangered {
};
```

Panda 通过 Raccoon 和 Bear 继承了 ZooAnimal，因为 Raccoon 和 Bear 继承 ZooAnimal 的方式都是虚继承，所以在 Panda 中只有一个 ZooAnimal 基类部分。

支持向基类的常规类型转换

不论基类是不是虚基类，派生类对象都能被可访问基类的指针或引用操作。例如，下面这些从 Panda 向基类的类型转换都是合法的：

```
void dance(const Bear&);
void rummage(const Raccoon&);
ostream& operator<<(ostream&, const ZooAnimal&);
Panda ying_yang;
dance(ying_yang);              // 正确：把一个 Panda 对象当成 Bear 传递
rummage(ying_yang);            // 正确：把一个 Panda 对象当成 Raccoon 传递
cout << ying_yang;             // 正确：把一个 Panda 对象当成 ZooAnimal 传递
```

虚基类成员的可见性

因为在每个共享的虚基类中只有唯一一个共享的子对象，所以该基类的成员可以被直接访问，并且不会产生二义性。此外，如果虚基类的成员只被一条派生路径覆盖，则我们仍然可以直接访问这个被覆盖的成员。但是如果成员被多余一个基类覆盖，则一般情况下派生类必须为该成员自定义一个新的版本。

例如，假定类 B 定义了一个名为 x 的成员，D1 和 D2 都是从 B 虚继承得到的，D 继承了 D1 和 D2，则在 D 的作用域中，x 通过 D 的两个基类都是可见的。如果我们通过 D 的对象使用 x，有三种可能性：

- 如果在 D1 和 D2 中都没有 x 的定义，则 x 将被解析为 B 的成员，此时不存在二义性，一个 D 的对象只含有 x 的一个实例。
- 如果 x 是 B 的成员，同时是 D1 和 D2 中某一个的成员，则同样没有二义性，派生类的 x 比共享虚基类 B 的 x 优先级更高。 813
- 如果在 D1 和 D2 中都有 x 的定义，则直接访问 x 将产生二义性问题。

与非虚的多重继承体系一样，解决这种二义性问题最好的方法是在派生类中为成员自定义新的实例。

18.3.4 节练习

练习 18.28：已知存在如下的继承体系，在 VMI 类的内部哪些继承而来的成员无须前缀限定符就能直接访问？哪些必须有限定符才能访问？说明你的原因。

```
struct Base {
    void bar(int);                  // 默认情况下是公有的
protected:
    int ival;
};
struct Derived1 : virtual public Base {
    void bar(char);                 // 默认情况下是公有的
    void foo(char);
protected:
    char cval;
};
struct Derived2 : virtual public Base {
    void foo(int);                  // 默认情况下是公有的
protected:
    int ival;
```

```
    char cval;
};
class VMI : public Derived1, public Derived2 { };
```

18.3.5 构造函数与虚继承

在虚派生中，虚基类是由最低层的派生类初始化的。以我们的程序为例，当创建 Panda 对象时，由 Panda 的构造函数独自控制 ZooAnimal 的初始化过程。

为了理解这一规则，我们不妨假设当以普通规则处理初始化任务时会发生什么情况。在此例中，虚基类将会在多条继承路径上被重复初始化。以 ZooAnimal 为例，如果应用普通规则，则 Raccoon 和 Bear 都会试图初始化 Panda 对象的 ZooAnimal 部分。

当然，继承体系中的每个类都可能在某个时刻成为“最低层的派生类”。只要我们能
814 创建虚基类的派生类对象，该派生类的构造函数就必须初始化它的虚基类。例如在我们的继承体系中，当创建一个 Bear（或 Raccoon）的对象时，它已经位于派生的最低层，因此 Bear（或 Raccoon）的构造函数将直接初始化其 ZooAnimal 基类部分：

```
Bear::Bear(std::string name, bool onExhibit):
          ZooAnimal(name, onExhibit, "Bear") { }
Raccoon::Raccoon(std::string name, bool onExhibit)
        : ZooAnimal(name, onExhibit, "Raccoon") { }
```

而当创建一个 Panda 对象时，Panda 位于派生的最低层并由它负责初始化共享的 ZooAnimal 基类部分。即使 ZooAnimal 不是 Panda 的直接基类，Panda 的构造函数也可以初始化 ZooAnimal：

```
Panda::Panda(std::string name, bool onExhibit)
      : ZooAnimal(name, onExhibit, "Panda"),
        Bear(name, onExhibit),
        Raccoon(name, onExhibit),
        Endangered(Endangered::critical),
        sleeping_flag(false) { }
```

虚继承的对象的构造方式

含有虚基类的对象的构造顺序与一般的顺序稍有区别：首先使用提供给最低层派生类构造函数的初始值初始化该对象的虚基类子部分，接下来按照直接基类在派生列表中出现的次序依次对其进行初始化。

例如，当我们创建一个 Panda 对象时：

- 首先使用 Panda 的构造函数初始值列表中提供的初始值构造虚基类 ZooAnimal 部分。
- 接下来构造 Bear 部分。
- 然后构造 Raccoon 部分。
- 然后构造第三个直接基类 Endangered。
- 最后构造 Panda 部分。

如果 Panda 没有显式地初始化 ZooAnimal 基类，则 ZooAnimal 的默认构造函数将被调用。如果 ZooAnimal 没有默认构造函数，则代码将发生错误。



虚基类总是先于非虚基类构造，与它们在继承体系中的次序和位置无关。

构造函数与析构函数的次序

一个类可以有多个虚基类。此时，这些虚的子对象按照它们在派生列表中出现的顺序从左向右依次构造。例如，在下面这个稍显杂乱的 TeddyBear 派生关系中有两个虚基类：ToyAnimal 是直接虚基类，ZooAnimal 是 Bear 的虚基类：

815

```
class Character { /* ...*/ };
class BookCharacter : public Character { /* ...*/ };
class ToyAnimal { /* ...*/ };
class TeddyBear : public BookCharacter,
                  public Bear, public virtual ToyAnimal
                  { /* ...*/ };
```

编译器按照直接基类的声明顺序对其依次进行检查，以确定其中是否含有虚基类。如果有，则先构造虚基类，然后按照声明的顺序逐一构造其他非虚基类。因此，要想创建一个 TeddyBear 对象，需要按照如下次序调用这些构造函数：

```
ZooAnimal();                    // Bear的虚基类
ToyAnimal();                    // 直接虚基类
Character();                    // 第一个非虚基类的间接基类
BookCharacter();                // 第一个直接非虚基类
Bear();                         // 第二个直接非虚基类
TeddyBear();                    // 最低层的派生类
```

合成的拷贝和移动构造函数按照完全相同的顺序执行，合成的赋值运算符中的成员也按照该顺序赋值。和往常一样，对象的销毁顺序与构造顺序正好相反，首先销毁 TeddyBear 部分，最后销毁 ZooAnimal 部分。

18.3.5 节练习

练习 18.29：已知有如下所示的类继承关系：

```
class Class { ... };
class Base : public Class { ... };
class D1 : virtual public Base { ... };
class D2 : virtual public Base { ... };
class MI : public D1, public D2 { ... };
class Final : public MI, public Class { ... };
```

(a)当作用于一个 Final 对象时，构造函数和析构函数的执行次序分别是什么？
(b)在一个 Final 对象中有几个 Base 部分？几个 Class 部分？
(c)下面的哪些赋值运算将造成编译错误？

```
Base *pb;       Class *pc;       MI *pmi;       D2 *pd2;
(a) pb = new Class;              (b) pc = new Final;
(c) pmi = pb;                    (d) pd2 = pmi;
```

练习 18.30：在 Base 中定义一个默认构造函数、一个拷贝构造函数和一个接受 int 形参的构造函数。在每个派生类中分别定义这三种构造函数，每个构造函数应该使用它的实参初始化其 Base 部分。

816 小结

C++语言可以用于解决各种类型的问题，既有几个小时就可以解决的小问题，也有一个大团队工作数年才能解决的超大规模问题。C++的某些特性特别适合于处理超大规模问题，这些特性包括：异常处理、命名空间以及多重继承或虚继承。

异常处理使得我们可以将程序的错误检测部分与错误处理部分分隔开来。当程序抛出一个异常时，当前正在执行的函数暂时中止，开始查找最邻近的与异常匹配的 catch 语句。作为异常处理的一部分，如果查找 catch 语句的过程中退出了某些函数，则函数中定义的局部变量也随之销毁。

命名空间是一种管理大规模复杂应用程序的机制，这些应用可能是由多个独立的供应商分别编写的代码组合而成的。一个命名空间是一个作用域，我们可以在其中定义对象、类型、函数、模板以及其他命名空间。标准库定义在名为 std 的命名空间中。

从概念上来说，多重继承非常简单：一个派生类可以从多个直接基类继承而来。在派生类对象中既包含派生类部分，也包含与每个基类对应的基类部分。虽然看起来很简单，但实际上多重继承的细节非常复杂。特别是对多个基类的继承可能会引入新的名字冲突，并造成来自于基类部分的名字的二义性问题。

如果一个类是从多个基类直接继承而来的，那么有可能这些基类本身又共享了另一个基类。在这种情况下，中间类可以选择使用虚继承，从而声明愿意与层次中虚继承同一基类的其他类共享虚基类。用这种方法，后代派生类中将只有一个共享虚基类的副本。

术语表

捕获所有异常（catch-all）异常声明形如 (...) 的 catch 子句。一条捕获所有异常的子句可以捕获任意类型的异常。常用于捕获局部检测的异常，该异常将重新抛出到程序的其他部分并最终解决问题。

catch 子句（catch clause） 程序中负责处理异常的部分。catch 子句包含关键字 catch，后面是异常声明以及一个语句块。catch 子句的代码负责处理异常声明中定义的异常。

构造函数顺序（constructor order） 在非虚继承中，基类的构造顺序与其在派生列表中出现的顺序一致。在虚继承中，首先构造虚基类。虚基类的构造顺序与其在派生类的派生列表中出现的顺序一致。只有最低层的派生类才能初始化虚基类。虚基类的初始值如果出现在中间基类中，则这些初始值将被忽略。

异常声明（exception declaration） catch 子句中指定其能够处理的异常类型的部分。异常声明的行为与形参列表类似，其中的唯一一个形参通过异常对象进行初始化。817 如果异常说明符是非引用类型，则异常对象将被拷贝给 catch。

异常处理（exception handling） 管理运行时异常的语言级支持。代码中一个独立开发的部分可以检测并引发异常，由程序的另一个独立开发的部分处理该异常。也就是说，程序的错误检测部分负责抛出异常，而错误处理部分在 try 语句块的 catch 子句中处理异常。

异常对象（exception object） 用于在异常的 throw 和 catch 之间进行通信的对象。在抛出点创建该对象，该对象是被抛出的表达式的副本。在该异常的最后一段处理代码完成之前异常对象都一直存在。异常对象的类型是被抛出的表达式的静态类型。

文件中的静态声明（file static） 使用关键字 static 声明的仅对当前文件有效的名字。在 C 语言和之前的 C++版本中，文件中的静态声明用于声明只能在当前文件中使用的名字。该特性在当前的 C++版本中已经被未命名的命名空间替换了。

函数 try 语句块（function try block） 用于捕获构造函数初始化过程发生的异常。关键字 try 出现在表示构造函数初始值列表开始的冒号之前（或者当初始值列表为空时出现在函数体的左侧花括号之前），并以函数体右侧花括号之后的一个或几个 catch 子句作为结束。

全局命名空间（global namespace） 是每个程序的隐式命名空间，用于存放全局定义。

处理代码（handler） 是“catch 子句”的同义词。

内联的命名空间（inline namespace） 内联命名空间中的名字可以看成是外层命名空间的成员。

多重继承（multiple inheritance） 有多个直接基类的类。派生类继承所有基类的成员。可以为每个基类分别设定访问说明符。

命名空间（namespace） 将库或者其他程序集定义的名字放在同一个作用域中的机制。和 C++的其他作用域不同，命名空间作用域可以定义成几个部分。我们可以打开并关闭命名空间，然后在程序的另一个地方重新打开并关闭该命名空间。

命名空间的别名（namespace alias） 为某个给定的命名空间定义同义词的机制：

```
namespace N1 = N;
```

将 N1 定义成命名空间 N 的另一个名字。命名空间可以含有多个别名，命名空间的原名和别名是等价的。

命名空间污染（namespace pollution） 当所有类和函数的名字都放置于全局命名空间时将造成命名空间污染。如果来自于多个独立供应商的代码都含有全局名字，则使用这些代码的大型程序很可能会面临命名空间污染的问题。

noexcept 运算符（noexcept operator） 该运算符返回一个 bool 值，用于表示给定的表达式是否会抛出异常。该表达式不会被求值，运算的结果是一个常量表达式。当提供的表达式不含 throw 并且只调用了做出不抛出说明的函数时，结果为 true；否则结果为 false。

noexcept 说明（noexcept specification） 表示函数是否会抛出异常的关键字。当 noexcept 跟在函数的形参列表之后时，它可以连接一个括号括起来的常量表达式，前提是该表达式可以转换成 bool 值。如果忽略了该表达式，或者表达式的值为 true，则函数不会抛出异常。如果表达式的值是 false 或者函数没有异常声明，则其可能抛出异常。

不抛出说明（nonthrowing specification） 该异常说明用于承诺某个函数不会抛出异常。如果一个做了不抛出说明的函数实际抛出了异常，将调用 terminate。不抛出说明符是不含实参或者含有一个值为 true 的实参的 noexcept。 818

引发（raise） 常常作为抛出的同义词。C++程序员认为抛出异常和引发异常基本上是等价的。

重新抛出（rethrow） 不指定表达式的 throw。重新抛出只有在 catch 子句内部或者被 catch 直接或间接调用了的函数内时才有效。它的效果是将其接受的异常重新抛出。

栈展开（stack unwinding） 在搜寻 catch 时依次退出函数的过程。异常发生前构造的局部对象将在进入相应的 catch 前被销毁。

terminate 是一个标准库函数，当异常未被捕获或者在处理异常的过程中发生了另一个异常时，terminate 负责结束程序的执行。

throw e 该表达式将中断当前的执行路径，throw 语句将控制权传递给最近的能够处

理该异常的 catch 子句。表达式 e 将被拷贝给异常对象。

try 语句块（try block） 含有关键字 try 以及一个或多个 catch 子句的语句块。如果 try 语句块中的代码引发了一个异常，并且某个 catch 可以匹配该异常，则异常将被这个 catch 处理。否则，异常被传递到 try 语句块之外并继续沿着调用链寻找与之匹配的 catch。

未命名的命名空间（unnamed namespace） 定义时未指定名字的命名空间。对于定义在未命名的命名空间中的名字，我们可以不用作用域运算符就直接访问它们。每个文件有一个独有的未命名的命名空间，其中的名字在文件外不可见。

using 声明（using declaration） 是一种将命名空间中的某个名字注入当前作用域的机制：

```
using std::cout;
```

上述语句使得命名空间 std 中的名字 cout 在当前作用域可见。之后，我们就可以直接使用 cout 而无须前缀 std::了。

using 指示（using directive） 是具有如下形式的声明：

```
using NS;
```

上述语句使得命名空间 NS 的所有名字在 using 指示所在的作用域以及 NS 所在的作用域都变得可见。

虚基类（virtual base class） 在派生列表中使用了关键字 virtual 的基类。在派生类对象中，虚基类部分只有一份，即使该虚基类在继承体系中出现了多次也是如此。对于非虚继承而言，构造函数只能初始化它的直接基类。但是对于虚继承来说，虚基类将被最低层的派生类初始化，因此最低层的派生类应该含有它的所有虚基类的初始值。

虚继承（virtual inheritance） 是多重继承的一种形式，基类被继承了多次，但是派生类共享该基类的唯一一份副本。

作用域运算符（:: operator） 用于访问命名空间或类中的名字。

第 19 章
特殊工具与技术

内容

19.1 控制内存分配 726
19.2 运行时类型识别 730
19.3 枚举类型 736
19.4 类成员指针 739
19.5 嵌套类 746
19.6 union：一种节省空间的类 749
19.7 局部类 754
19.8 固有的不可移植的特性 755
小结 762
术语表 762

本书的前三部分讨论了 C++语言的基本要素，这些要素绝大多数程序员都会用到。此外，C++还定义了一些非常特殊的性质，对于很多程序员来说，他们一般很少会用到本章介绍的内容。

820 C++语言的设计者希望它能处理各种各样的问题。因此，C++的某些特征可能对于一些特殊的应用非常重要，而在另外一些情况下没什么作用。本章将介绍C++语言的几种未被广泛使用的特征。

19.1 控制内存分配

某些应用程序对内存分配有特殊的需求，因此我们无法将标准内存管理机制直接应用于这些程序。它们常常需要自定义内存分配的细节，比如使用关键字 new 将对象放置在特定的内存空间中。为了实现这一目的，应用程序需要重载 new 运算符和 delete 运算符以控制内存分配的过程。

19.1.1 重载 new 和 delete

尽管我们说能够“重载 new 和 delete”，但是实际上重载这两个运算符与重载其他运算符的过程大不相同。要想真正掌握重载 new 和 delete 的方法，首先要对 new 表达式和 delete 表达式的工作机理有更多了解。

当我们使用一条 new 表达式时：

```
// new 表达式
string *sp = new string("a value");   // 分配并初始化一个 string 对象
string *arr = new string[10];         // 分配 10 个默认初始化的 string 对象
```

实际执行了三步操作。第一步，new 表达式调用一个名为 **operator new**（或者 **operator new[]**）的标准库函数。该函数分配一块足够大的、原始的、未命名的内存空间以便存储特定类型的对象（或者对象的数组）。第二步，编译器运行相应的构造函数以构造这些对象，并为其传入初始值。第三步，对象被分配了空间并构造完成，返回一个指向该对象的指针。

当我们使用一条 delete 表达式删除一个动态分配的对象时：

```
delete sp;                    // 销毁*sp，然后释放 sp 指向的内存空间
delete [] arr;                // 销毁数组中的元素，然后释放对应的内存空间
```

实际执行了两步操作。第一步，对 sp 所指的对象或者 arr 所指的数组中的元素执行对应的析构函数。第二步，编译器调用名为 **operator delete**（或者 **operator delete[ ]**）的标准库函数释放内存空间。

如果应用程序希望控制内存分配的过程，则它们需要定义自己的 operator new 函数和 operator delete 函数。即使在标准库中已经存在这两个函数的定义，我们仍旧可以定义自己的版本。编译器不会对这种重复的定义提出异议，相反，编译器将使用我们自定义的版本替换标准库定义的版本。

821



当自定义了全局的 operator new 函数和 operator delete 函数后，我们就担负起了控制动态内存分配的职责。这两个函数必须是正确的：因为它们是程序整个处理过程中至关重要的一部分。

应用程序可以在全局作用域中定义 operator new 函数和 operator delete 函数，也可以将它们定义为成员函数。当编译器发现一条 new 表达式或 delete 表达式后，将

在程序中查找可供调用的 operator 函数。如果被分配（释放）的对象是类类型，则编译器首先在类及其基类的作用域中查找。此时如果该类含有 operator new 成员或 operator delete 成员，则相应的表达式将调用这些成员。否则，编译器在全局作用域查找匹配的函数。此时如果编译器找到了用户自定义的版本，则使用该版本执行 new 表达式或 delete 表达式；如果没找到，则使用标准库定义的版本。

我们可以使用作用域运算符令 new 表达式或 delete 表达式忽略定义在类中的函数，直接执行全局作用域中的版本。例如，::new 只在全局作用域中查找匹配的 operator new 函数，::delete 与之类似。

operator new 接口和 operator delete 接口

标准库定义了 operator new 函数和 operator delete 函数的 8 个重载版本。其中前 4 个版本可能抛出 bad_alloc 异常，后 4 个版本则不会抛出异常：

```
// 这些版本可能抛出异常
void *operator new(size_t);                    // 分配一个对象
void *operator new[](size_t);                  // 分配一个数组
void *operator delete(void*) noexcept;         // 释放一个对象
void *operator delete[](void*) noexcept;       // 释放一个数组

// 这些版本承诺不会抛出异常，参见 12.1.2 节（第 409 页）
void *operator new(size_t, nothrow_t&) noexcept;
void *operator new[](size_t, nothrow_t&) noexcept;
void *operator delete(void*, nothrow_t&) noexcept;
void *operator delete[](void*, nothrow_t&) noexcept;
```

类型 nothrow_t 是定义在 new 头文件中的一个 struct，在这个类型中不包含任何成员。new 头文件还定义了一个名为 nothrow 的 const 对象，用户可以通过这个对象请求 new 的非抛出版本（参见 12.1.2 节，第 408 页）。与析构函数类似，operator delete 也不允许抛出异常（参见 18.1.1 节，第 685 页）。当我们重载这些运算符时，必须使用 noexcept 异常说明符（参见 18.1.4 节，第 690 页）指定其不抛出异常。

应用程序可以自定义上面函数版本中的任意一个，前提是自定义的版本必须位于全局作用域或者类作用域中。当我们将上述运算符函数定义成类的成员时，它们是隐式静态的（参见 7.6 节，第 270 页）。我们无须显式地声明 static，当然这么做也不会引发错误。因为 operator new 用在对象构造之前而 operator delete 用在对象销毁之后，所以这两个成员（new 和 delete）必须是静态的，而且它们不能操纵类的任何数据成员。 822

对于 operator new 函数或者 operator new[]函数来说，它的返回类型必须是 void*，第一个形参的类型必须是 size_t 且该形参不能含有默认实参。当我们为一个对象分配空间时使用 operator new；为一个数组分配空间时使用 operator new[]。当编译器调用 operator new 时，把存储指定类型对象所需的字节数传给 size_t 形参；当调用 operator new[]时，传入函数的则是存储数组中所有元素所需的空间。

如果我们想要自定义 operator new 函数，则可以为它提供额外的形参。此时，用到这些自定义函数的 new 表达式必须使用 new 的定位形式（参见 12.1.2 节，第 409 页）将实参传给新增的形参。尽管在一般情况下我们可以自定义具有任何形参的 operator new，但是下面这个函数却无论如何不能被用户重载：

```
void *operator new(size_t, void*);             // 不允许重新定义这个版本
```

这种形式只供标准库使用，不能被用户重新定义。

对于 operator delete 函数或者 operator delete[]函数来说，它们的返回类型必须是 void，第一个形参的类型必须是 void*。执行一条 delete 表达式将调用相应的 operator 函数，并用指向待释放内存的指针来初始化 void*形参。

当我们将 operator delete 或 operator delete[]定义成类的成员时，该函数可以包含另外一个类型为 size_t 的形参。此时，该形参的初始值是第一个形参所指对象的字节数。size_t 形参可用于删除继承体系中的对象。如果基类有一个虚析构函数（参见 15.7.1 节，第 552 页），则传递给 operator delete 的字节数将因待删除指针所指对象的动态类型不同而有所区别。而且，实际运行的 operator delete 函数版本也由对象的动态类型决定。

术语：new 表达式与 operator new 函数

标准库函数 operator new 和 operator delete 的名字容易让人误解。和其他 operator 函数不同(比如 operator=),这两个函数并没有重载 new 表达式或 delete 表达式。实际上，我们根本无法自定义 new 表达式或 delete 表达式的行为。

一条 new 表达式的执行过程总是先调用 operator new 函数以获取内存空间，然后在得到的内存空间中构造对象。与之相反，一条 delete 表达式的执行过程总是先销毁对象，然后调用 operator delete 函数释放对象所占的空间。

我们提供新的 operator new 函数和 operator delete 函数的目的在于改变内存分配的方式，但是不管怎样，我们都不能改变 new 运算符和 delete 运算符的基本含义。

823 > malloc 函数与 free 函数

当你定义了自己的全局 operator new 和 operator delete 后，这两个函数必须以某种方式执行分配内存与释放内存的操作。也许你的初衷仅仅是使用一个特殊定制的内存分配器，但是这两个函数还应该同时满足某些测试的目的，即检验其分配内存的方式是否与常规方式类似。

为此，我们可以使用名为 **malloc** 和 **free** 的函数，C++从 C 语言中继承了这些函数，并将其定义在 cstdlib 头文件中。

malloc 函数接受一个表示待分配字节数的 size_t，返回指向分配空间的指针或者返回 0 以表示分配失败。free 函数接受一个 void*，它是 malloc 返回的指针的副本，free 将相关内存返回给系统。调用 free(0)没有任何意义。

如下所示是编写 operator new 和 operator delete 的一种简单方式，其他版本与之类似：

```
void *operator new(size_t size) {
    if (void *mem = malloc(size))
        return mem;
    else
        throw bad_alloc();
}
void operator delete(void *mem) noexcept { free(mem); }
```

19.1.1 节练习

练习 19.1：使用 malloc 编写你自己的 operator new(size_t) 函数，使用 free 编写 operator delete(void *) 函数。

练习 19.2：默认情况下，allocator 类使用 operator new 获取存储空间，然后使用 operator delete 释放它。利用上一题中的两个函数重新编译并运行你的 StrVec 程序（参见 13.5 节，第 465 页）。

19.1.2 定位 new 表达式

尽管 operator new 函数和 operator delete 函数一般用于 new 表达式，然而它们毕竟是标准库的两个普通函数，因此普通的代码也可以直接调用它们。

在 C++的早期版本中，allocator 类（参见 12.2.2 节，第 427 页）还不是标准库的一部分。应用程序如果想把内存分配与初始化分离开来的话，需要调用 operator new 和 operator delete。这两个函数的行为与 allocator 的 allocate 成员和 deallocate 成员非常类似，它们负责分配或释放内存空间，但是不会构造或销毁对象。

与 allocator 不同的是，对于 operator new 分配的内存空间来说我们无法使用 construct 函数构造对象。相反，我们应该使用 new 的**定位 new**（placement new）形式（参见 12.1.2 节，第 409 页）构造对象。如我们所知，new 的这种形式为分配函数提供了额外的信息。我们可以使用定位 new 传递一个地址，此时定位 new 的形式如下所示： 824

```
new (place_address) type
new (place_address) type (initializers)
new (place_address) type [size]
new (place_address) type [size] { braced initializer list }
```

其中 *place_address* 必须是一个指针，同时在 *initializers* 中提供一个（可能为空的）以逗号分隔的初始值列表，该初始值列表将用于构造新分配的对象。

当仅通过一个地址值调用时，定位 new 使用 operator new(size_t, void*)“分配”它的内存。这是一个我们无法自定义的 operator new 版本（参见 19.1.1 节，第 727 页）。该函数不分配任何内存，它只是简单地返回指针实参；然后由 new 表达式负责在指定的地址初始化对象以完成整个工作。事实上，定位 new 允许我们在一个特定的、预先分配的内存地址上构造对象。



当只传入一个指针类型的实参时，定位 new 表达式构造对象但是不分配内存。

尽管在很多时候使用定位 new 与 allocator 的 construct 成员非常相似，但在它们之间也有一个重要的区别。我们传给 construct 的指针必须指向同一个 allocator 对象分配的空间，但是传给定位 new 的指针无须指向 operator new 分配的内存。实际上如我们将在 19.6 节（第 753 页）介绍的，传给定位 new 表达式的指针甚至不需要指向动态内存。

显式的析构函数调用

就像定位 new 与使用 allocate 类似一样，对析构函数的显式调用也与使用 destroy 很类似。我们既可以通过对象调用析构函数，也可以通过对象的指针或引用调

用析构函数，这与调用其他成员函数没什么区别：

```
string *sp = new string("a value");   // 分配并初始化一个 string 对象
sp->~string();
```

在这里我们直接调用了一个析构函数。箭头运算符解引用指针 sp 以获得 sp 所指的对象，然后我们调用析构函数，析构函数的形式是波浪线（~）加上类型的名字。

和调用 destroy 类似，调用析构函数可以清除给定的对象但是不会释放该对象所在的空间。如果需要的话，我们可以重新使用该空间。



调用析构函数会销毁对象，但是不会释放内存。

825

19.2 运行时类型识别

运行时类型识别（run-time type identification，RTTI）的功能由两个运算符实现：

- typeid 运算符，用于返回表达式的类型。
- dynamic_cast 运算符，用于将基类的指针或引用安全地转换成派生类的指针或引用。

当我们将这两个运算符用于某种类型的指针或引用，并且该类型含有虚函数时，运算符将使用指针或引用所绑定对象的动态类型（参见 15.2.3 节，第 534 页）。

这两个运算符特别适用于以下情况：我们想使用基类对象的指针或引用执行某个派生类操作并且该操作不是虚函数。一般来说，只要有可能我们应该尽量使用虚函数。当操作被定义成虚函数时，编译器将根据对象的动态类型自动地选择正确的函数版本。

然而，并非任何时候都能定义一个虚函数。假设我们无法使用虚函数，则可以使用一个 RTTI 运算符。另一方面，与虚成员函数相比，使用 RTTI 运算符蕴含着更多潜在的风险：程序员必须清楚地知道转换的目标类型并且必须检查类型转换是否被成功执行。



使用 RTTI 必须要加倍小心。在可能的情况下，最好定义虚函数而非直接接管类型管理的重任。

19.2.1 dynamic_cast 运算符

dynamic_cast 运算符（dynamic_cast operator）的使用形式如下所示：

```
dynamic_cast<type*>(e)
dynamic_cast<type&>(e)
dynamic_cast<type&&>(e)
```

其中，*type* 必须是一个类类型，并且通常情况下该类型应该含有虚函数。在第一种形式中，e 必须是一个有效的指针（参见 2.3.2 节，第 47 页）；在第二种形式中，e 必须是一个左值；在第三种形式中，e 不能是左值。

在上面的所有形式中，e 的类型必须符合以下三个条件中的任意一个：e 的类型是目标 *type* 的公有派生类、e 的类型是目标 *type* 的公有基类或者 e 的类型就是目标 *type* 的类型。如果符合，则类型转换可以成功。否则，转换失败。如果一条 dynamic_cast 语句的转换目标是指针类型并且失败了，则结果为 0。如果转换目标是引用类型并且失败了，

则 dynamic_cast 运算符将抛出一个 bad_cast 异常。

指针类型的 dynamic_cast

举个简单的例子，假定 Base 类至少含有一个虚函数，Derived 是 Base 的公有派生类。如果有一个指向 Base 的指针 bp，则我们可以在运行时将它转换成指向 Derived 的指针，具体代码如下：

826

```
if (Derived *dp = dynamic_cast<Derived*>(bp))
{
    // 使用 dp 指向的 Derived 对象
} else { // bp 指向一个 Base 对象
    // 使用 bp 指向的 Base 对象
}
```

如果 bp 指向 Derived 对象，则上述的类型转换初始化 dp 并令其指向 bp 所指的 Derived 对象。此时，if 语句内部使用 Derived 操作的代码是安全的。否则，类型转换的结果为 0，dp 为 0 意味着 if 语句的条件失败，此时 else 子句执行相应的 Base 操作。



我们可以对一个空指针执行 dynamic_cast，结果是所需类型的空指针。

值得注意的一点是，我们在条件部分定义了 dp，这样做的好处是可以在一个操作中同时完成类型转换和条件检查两项任务。而且，指针 dp 在 if 语句外部是不可访问的。一旦转换失败，即使后续的代码忘了做相应判断，也不会接触到这个未绑定的指针，从而确保程序是安全的。



在条件部分执行 dynamic_cast 操作可以确保类型转换和结果检查在同一条表达式中完成。

引用类型的 dynamic_cast

引用类型的 dynamic_cast 与指针类型的 dynamic_cast 在表示错误发生的方式上略有不同。因为不存在所谓的空引用，所以对于引用类型来说无法使用与指针类型完全相同的错误报告策略。当对引用的类型转换失败时，程序抛出一个名为 std::bad_cast 的异常，该异常定义在 typeinfo 标准库头文件中。

我们可以按照如下的形式改写之前的程序，令其使用引用类型：

```
void f(const Base &b)
{
    try {
        const Derived &d = dynamic_cast<const Derived&>(b);
    // 使用 b 引用的 Derived 对象
    } catch (bad_cast) {
        // 处理类型转换失败的情况
    }
}
```

19.2.1 节练习

练习 19.3：已知存在如下的类继承体系，其中每个类分别定义了一个公有的默认构造函

数和一个虚析构函数：

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public B { /* ... */ };
class D : public B, public A { /* ... */ };
```

下面的哪个 dynamic_cast 将失败？

```
(a) A *pa = new C;
    B *pb = dynamic_cast< B* >(pa);
(b) B *pb = new B;
    C *pc = dynamic_cast< C* >(pb);
(c) A *pa = new D;
    B *pb = dynamic_cast< B* >(pa);
```

练习 19.4：使用上一个练习定义的类改写下面的代码，将表达式*pa 转换成类型 C&：

```
if (C *pc = dynamic_cast< C* >(pa))
}
    // 使用 C 的成员
} else {
    // 使用 A 的成员
}
```

练习 19.5：在什么情况下你应该使用 dynamic_cast 替代虚函数？

19.2.2 typeid 运算符

为 RTTI 提供的第二个运算符是 **typeid 运算符**（typeid operator），它允许程序向表达式提问：你的对象是什么类型？

827 typeid 表达式的形式是 typeid(e)，其中 e 可以是任意表达式或类型的名字。typeid 操作的结果是一个常量对象的引用，该对象的类型是标准库类型 type_info 或者 type_info 的公有派生类型。type_info 类定义在 typeinfo 头文件中，19.2.4 节（第 735 页）将介绍更多关于 type_info 的细节。

typeid 运算符可以作用于任意类型的表达式。和往常一样，顶层 const（参见 2.4.3 节，第 57 页）被忽略，如果表达式是一个引用，则 typeid 返回该引用所引对象的类型。不过当 typeid 作用于数组或函数时，并不会执行向指针的标准类型转换（参见 4.11.2 节，第 143 页）。也就是说，如果我们对数组 a 执行 typeid(a)，则所得的结果是数组类型而非指针类型。

当运算对象不属于类类型或者是一个不包含任何虚函数的类时，typeid 运算符指示的是运算对象的静态类型。而当运算对象是定义了至少一个虚函数的类的左值时，typeid 的结果直到运行时才会求得。

使用 typeid 运算符

通常情况下，我们使用 typeid 比较两条表达式的类型是否相同，或者比较一条表达式的类型是否与指定类型相同：

828

```
Derived *dp = new Derived;
Base *bp = dp;                          // 两个指针都指向 Derived 对象
// 在运行时比较两个对象的类型
```

```
if (typeid(*bp) == typeid(*dp)) {
    // bp和dp指向同一类型的对象
}
// 检查运行时类型是否是某种指定的类型
if (typeid(*bp) == typeid(Derived)) {
    // bp实际指向Derived对象
}
```

在第一个 if 语句中，我们比较 bp 和 dp 所指的对象的动态类型是否相同。如果相同，则条件成功。类似的，当 bp 当前所指的是一个 Derived 对象时，第二个 if 语句的条件满足。

注意，typeid 应该作用于对象，因此我们使用*bp 而非 bp：

```
// 下面的检查永远是失败的：bp的类型是指向Base的指针
if (typeid(bp) == typeid(Derived)) {
    // 此处的代码永远不会执行
}
```

这个条件比较的是类型 Base*和 Derived。尽管指针所指的对象类型是一个含有虚函数的类，但是指针本身并不是一个类类型的对象。类型 Base*将在编译时求值，显然它与 Derived 不同，因此不论 bp 所指的对象到底是什么类型，上面的条件都不会满足。



当 typeid 作用于指针时（而非指针所指的对象），返回的结果是该指针的静态编译时类型。

typeid 是否需要运行时检查决定了表达式是否会被求值。只有当类型含有虚函数时，编译器才会对表达式求值。反之，如果类型不含有虚函数，则 typeid 返回表达式的静态类型；编译器无须对表达式求值也能知道表达式的静态类型。

如果表达式的动态类型可能与静态类型不同，则必须在运行时对表达式求值以确定返回的类型。这条规则适用于 typeid(*p) 的情况。如果指针 p 所指的类型不含有虚函数，则 p 不必非得是一个有效的指针。否则，*p 将在运行时求值，此时 p 必须是一个有效的指针。如果 p 是一个空指针，则 typeid(*p) 将抛出一个名为 bad_typeid 的异常。

19.2.2 节练习

练习 19.6：编写一条表达式将 Query_base 指针动态转换为 AndQuery 指针（参见 15.9.1 节，第 564 页）。分别使用 AndQuery 的对象以及其他类型的对象测试转换是否有效。打印一条表示类型转换是否成功的信息，确保实际输出的结果与期望的一致。

练习 19.7：编写与上一个练习类似的转换，这一次将 Query_base 对象转换为 AndQuery 的引用。重复上面的测试过程，确保转换能正常工作。

练习 19.8：编写一条 typeid 表达式检查两个 Query_base 对象是否指向同一种类型。再检查该类型是否是 AndQuery。

19.2.3 使用 RTTI

在某些情况下 RTTI 非常有用，比如当我们想为具有继承关系的类实现相等运算符时（参见 14.3.1 节，第 497 页）。对于两个对象来说，如果它们的类型相同并且对应的数据成

员取值相同，则我们说这两个对象是相等的。在类的继承体系中，每个派生类负责添加自己的数据成员，因此派生类的相等运算符必须把派生类的新成员考虑进来。

829 一种容易想到的解决方案是定义一套虚函数，令其在继承体系的各个层次上分别执行相等性判断。此时，我们可以为基类的引用定义一个相等运算符，该运算符将它的工作委托给虚函数 equal，由 equal 负责实际的操作。

遗憾的是，上述方案很难奏效。虚函数的基类版本和派生类版本必须具有相同的形参类型（参见 15.3 节，第 537 页）。如果我们想定义一个虚函数 equal，则该函数的形参必须是基类的引用。此时，equal 函数将只能使用基类的成员，而不能比较派生类独有的成员。

要想实现真正有效的相等比较操作，我们需要首先清楚一个事实：即如果参与比较的两个对象类型不同，则比较结果为 false。例如，如果我们试图比较一个基类对象和一个派生类对象，则==运算符应该返回 false。

基于上述推论，我们就可以使用 RTTI 解决问题了。我们定义的相等运算符的形参是基类的引用，然后使用 typeid 检查两个运算对象的类型是否一致。如果运算对象的类型不一致，则==返回 false；类型一致才调用 equal 函数。每个类定义的 equal 函数负责比较类型自己的成员。这些运算符接受 Base&形参，但是在进行比较操作前先把运算对象转换成运算符所属的类类型。

类的层次关系

为了更好地解释上述概念，我们定义两个示例类：

```
class Base {
    friend bool operator==(const Base&, const Base&);
public:
    // Base 的接口成员
protected:
    virtual bool equal(const Base&) const;
    // Base 的数据成员和其他用于实现的成员
};
```

830

```
class Derived: public Base {
public:
    // Derived 的其他接口成员
protected:
    bool equal(const Base&) const;
    // Derived 的数据成员和其他用于实现的成员
};
```

类型敏感的相等运算符

接下来介绍我们是如何定义整体的相等运算符的：

```
bool operator==(const Base &lhs, const Base &rhs)
{
    // 如果 typeid 不相同，返回 false；否则虚调用 equal
    return typeid(lhs) == typeid(rhs) && lhs.equal(rhs);
}
```

在这个运算符中，如果运算对象的类型不同则返回 false。否则，如果运算对象的类型相同，则运算符将其工作委托给虚函数 equal。当运算对象是 Base 的对象时，调用 Base::equal；当运算对象是 Derived 的对象时，调用 Derived::equal。

虚 equal 函数

继承体系中的每个类必须定义自己的 equal 函数。派生类的所有函数要做的第一件事都是相同的，那就是将实参的类型转换为派生类类型：

```
bool Derived::equal(const Base &rhs) const
{
    // 我们清楚这两个类型是相等的，所以转换过程不会抛出异常
    auto r = dynamic_cast<const Derived&>(rhs);
    // 执行比较两个 Derived 对象的操作并返回结果
}
```

上面的类型转换永远不会失败，因为毕竟我们只有在验证了运算对象的类型相同之后才会调用该函数。然而这样的类型转换是必不可少的，执行了类型转换后，当前函数才能访问右侧运算对象的派生类成员。

基类 equal 函数

下面这个操作比其他的稍微简单一点：

```
bool Base::equal(const Base &rhs) const
{
    // 执行比较 Base 对象的操作
}
```

无须事先转换形参的类型。*this 和形参都是 Base 对象，因此当前对象可用的操作对于形参类型同样有效。

19.2.4 type_info 类

831

type_info 类的精确定义随着编译器的不同而略有差异。不过，C++标准规定 type_info 类必须定义在 typeinfo 头文件中，并且至少提供表 19.1 所列的操作。

表 19.1：type_info 的操作

t1 == t2	如果 type_info 对象 t1 和 t2 表示同一种类型，返回 true；否则返回 false
t1 != t2	如果 type_info 对象 t1 和 t2 表示不同的类型，返回 true；否则返回 false
t.name()	返回一个 C 风格字符串，表示类型名字的可打印形式。类型名字的生成方式因系统而异
t1.before(t2)	返回一个 bool 值，表示 t1 是否位于 t2 之前。before 所采用的顺序关系是依赖于编译器的

除此之外，因为 type_info 类一般是作为一个基类出现，所以它还应该提供一个公有的虚析构函数。当编译器希望提供额外的类型信息时，通常在 type_info 的派生类中完成。

type_info 类没有默认构造函数，而且它的拷贝和移动构造函数以及赋值运算符都被定义成删除的（参见 13.1.6 节，第 450 页）。因此，我们无法定义或拷贝 type_info 类型的对象，也不能为 type_info 类型的对象赋值。创建 type_info 对象的唯一途径是使用 typeid 运算符。

type_info 类的 name 成员函数返回一个 C 风格字符串，表示对象的类型名字。对

于某种给定的类型来说，name 的返回值因编译器而异并且不一定与在程序中使用的名字一致。对于 name 返回值的唯一要求是，类型不同则返回的字符串必须有所区别。例如：

```
int arr[10];
Derived d;
Base *p = &d;

cout << typeid(42).name() << ", "
     << typeid(arr).name() << ", "
     << typeid(Sales_data).name() << ", "
     << typeid(std::string).name() << ", "
     << typeid(p).name() << ", "
     << typeid(*p).name() << endl;
```

在作者的计算机上运行该程序，输出结果如下：

```
i, A10_i, 10Sales_data, Ss, P4Base, 7Derived
```



type_info 类在不同的编译器上有所区别。有的编译器提供了额外的成员函数以提供程序中所用类型的额外信息。读者应该仔细阅读你所用编译器的使用手册，从而获取关于 type_info 的更多细节。

832

19.2.4 节练习

练习 19.9：编写与本节最后一个程序类似的代码，令其打印你的编译器为一些常见类型所起的名字。如果你得到的输出结果与本书类似，尝试编写一个函数将这些字符串翻译成人们更容易读懂的形式。

练习 19.10：已知存在如下的类继承体系，其中每个类定义了一个默认公有的构造函数和一个虚析构函数。下面的语句将打印哪些类型名字？

```
class A { /* ...*/ };
class B : public A { /* ...*/ };
class C : public B { /* ...*/ };

(a) A *pa = new C;
    cout << typeid(pa).name() << endl;
(b) C cobj;
    A& ra = cobj;
    cout << typeid(&ra).name() << endl;
(c) B *px = new B;
    A& ra = *px;
    cout << typeid(ra).name() << endl;
```

19.3 枚举类型

枚举类型（enumeration）使我们可以将一组整型常量组织在一起。和类一样，每个枚举类型定义了一种新的类型。枚举属于字面值常量类型（参见 7.5.6 节，第 267 页）。

C++包含两种枚举：限定作用域的和不限定作用域的。C++11 新标准引入了**限定作用域的枚举类型**（scoped enumeration）。定义限定作用域的枚举类型的一般形式是：首先是关键字 enum class（或者等价地使用 enum struct），随后是枚举类型名字以及用花括

C++ 11

号括起来的以逗号分隔的**枚举成员**（enumerator）列表，最后是一个分号：

```
enum class open_modes {input, output, append};
```

我们定义了一个名为 open_modes 的枚举类型，它包含三个枚举成员：input、output 和 append。

定义**不限定作用域的枚举类型**（unscoped enumeration）时省略掉关键字 class（或 struct），枚举类型的名字是可选的：

```
enum color {red, yellow, green};      // 不限定作用域的枚举类型
// 未命名的、不限定作用域的枚举类型
enum {floatPrec = 6, doublePrec = 10, double_doublePrec = 10};
```

如果 enum 是未命名的，则我们只能在定义该 enum 时定义它的对象。和类的定义类似，我们需要在 enum 定义的右侧花括号和最后的分号之间提供逗号分隔的声明列表（参见 2.6.1 节，第 64 页）。

枚举成员

833

在限定作用域的枚举类型中，枚举成员的名字遵循常规的作用域准则，并且在枚举类型的作用域外是不可访问的。与之相反，在不限定作用域的枚举类型中，枚举成员的作用域与枚举类型本身的作用域相同：

```
enum color {red, yellow, green};          // 不限定作用域的枚举类型
enum stoplight {red, yellow, green};      // 错误：重复定义了枚举成员
enum class peppers {red, yellow, green}; // 正确：枚举成员被隐藏了
color eyes = green; // 正确：不限定作用域的枚举类型的枚举成员位于有效的作用域中
peppers p = green;  // 错误：peppers 的枚举成员不在有效的作用域中
                    // color::green 在有效的作用域中，但是类型错误
color hair = color::red;                  // 正确：允许显式地访问枚举成员
peppers p2 = peppers::red;                // 正确：使用 pappers 的 red
```

默认情况下，枚举值从 0 开始，依次加 1。不过我们也能为一个或几个枚举成员指定专门的值：

```
enum class intTypes {
    charTyp = 8, shortTyp = 16, intTyp = 16,
    longTyp = 32, long_longTyp = 64
};
```

由枚举成员 intTyp 和 shortTyp 可知，枚举值不一定唯一。如果我们没有显式地提供初始值，则当前枚举成员的值等于之前枚举成员的值加 1。

枚举成员是 const，因此在初始化枚举成员时提供的初始值必须是常量表达式（参见 2.4.4 节，第 58 页）。也就是说，每个枚举成员本身就是一条常量表达式，我们可以在任何需要常量表达式的地方使用枚举成员。例如，我们可以定义枚举类型的 constexpr 变量：

```
constexpr intTypes charbits = intTypes::charTyp;
```

类似的，我们也可以将一个 enum 作为 switch 语句的表达式，而将枚举值作为 case 标签（参见 5.3.2 节，第 160 页）。出于同样的原因，我们还能将枚举类型作为一个非类型模板形参使用（参见 16.1.1 节，第 580 页）；或者在类的定义中初始化枚举类型的静态数据成员（参见 7.6 节，第 270 页）。

和类一样，枚举也定义新的类型

只要enum有名字，我们就能定义并初始化该类型的成员。要想初始化enum对象或者为enum对象赋值，必须使用该类型的一个枚举成员或者该类型的另一个对象：

```
open_modes om = 2;              // 错误：2不属于类型open_modes
om = open_modes::input;         // 正确：input是open_modes的一个枚举成员
```

834

一个不限定作用域的枚举类型的对象或枚举成员自动地转换成整型。因此，我们可以在任何需要整型值的地方使用它们：

```
int i = color::red;  // 正确：不限定作用域的枚举类型的枚举成员隐式地转换成int
int j = peppers::red;// 错误：限定作用域的枚举类型不会进行隐式转换
```

指定enum的大小

C++11

尽管每个enum都定义了唯一的类型，但实际上enum是由某种整数类型表示的。在C++11新标准中，我们可以在enum的名字后加上冒号以及我们想在该enum中使用的类型：

```
enum intValues : unsigned long long {
    charTyp = 255, shortTyp = 65535, intTyp = 65535,
    longTyp = 4294967295UL,
    long_longTyp = 18446744073709551615ULL
};
```

如果我们没有指定enum的潜在类型，则默认情况下限定作用域的enum成员类型是int。对于不限定作用域的枚举类型来说，其枚举成员不存在默认类型，我们只知道成员的潜在类型足够大，肯定能够容纳枚举值。如果我们指定了枚举成员的潜在类型（包括对限定作用域的enum的隐式指定），则一旦某个枚举成员的值超出了该类型所能容纳的范围，将引发程序错误。

指定enum潜在类型的能力使得我们可以控制不同实现环境中使用的类型，我们将可以确保在一种实现环境中编译通过的程序所生成的代码与其他实现环境中生成的代码一致。

枚举类型的前置声明

C++11

在C++11新标准中，我们可以提前声明enum。enum的前置声明（无论隐式地还是显示地）必须指定其成员的大小：

```
// 不限定作用域的枚举类型intValues的前置声明
enum intValues : unsigned long long;  // 不限定作用域的，必须指定成员类型
enum class open_modes;    // 限定作用域的枚举类型可以使用默认成员类型int
```

因为不限定作用域的enum未指定成员的默认大小，因此每个声明必须指定成员的大小。对于限定作用域的enum来说，我们可以不指定其成员的大小，这个值被隐式地定义成int。

和其他声明语句一样，enum的声明和定义必须匹配，这意味着在该enum的所有声明和定义中成员的大小必须一致。而且，我们不能在同一个上下文中先声明一个不限定作用域的enum名字，然后再声明一个同名的限定作用域的enum：

```
// 错误：所有的声明和定义必须对该enum是限定作用域的还是不限定作用域的保持一致
enum class intValues;
enum intValues;             // 错误：intValues已经被声明成限定作用域的enum
enum intValues : long;      // 错误：intValues已经被声明成int
```

835

形参匹配与枚举类型

要想初始化一个 enum 对象，必须使用该 enum 类型的另一个对象或者它的一个枚举成员（参见 19.3 节，第 737 页）。因此，即使某个整型值恰好与枚举成员的值相等，它也不能作为函数的 enum 实参使用：

```
// 不限定作用域的枚举类型，潜在类型因机器而异
enum Tokens {INLINE = 128, VIRTUAL = 129};
void ff(Tokens);
void ff(int);
int main() {
    Tokens curTok = INLINE;
    ff(128);                    // 精确匹配 ff(int)
    ff(INLINE);                 // 精确匹配 ff(Tokens)
    ff(curTok);                 // 精确匹配 ff(Tokens)
    return 0;
}
```

尽管我们不能直接将整型值传给 enum 形参，但是可以将一个不限定作用域的枚举类型的对象或枚举成员传给整型形参。此时，enum 的值提升成 int 或更大的整型，实际提升的结果由枚举类型的潜在类型决定：

```
void newf(unsigned char);
void newf(int);
unsigned char uc = VIRTUAL;
newf(VIRTUAL);                  // 调用 newf(int)
newf(uc);                       // 调用 newf(unsigned char)
```

枚举类型 Tokens 只有两个枚举成员，其中较大的那个值是 129。该枚举类型可以用 unsigned char 来表示，因此很多编译器使用 unsigned char 作为 Tokens 的潜在类型。不管 Tokens 的潜在类型到底是什么，它的对象和枚举成员都提升成 int。尤其是，枚举成员永远不会提升成 unsigned char，即使枚举值可以用 unsigned char 存储也是如此。

19.4 类成员指针

成员指针（pointer to member）是指可以指向类的非静态成员的指针。一般情况下，指针指向一个对象，但是成员指针指示的是类的成员，而非类的对象。类的静态成员不属于任何对象，因此无须特殊的指向静态成员的指针，指向静态成员的指针与普通指针没有什么区别。

成员指针的类型囊括了类的类型以及成员的类型。当初始化一个这样的指针时，我们令其指向类的某个成员，但是不指定该成员所属的对象；直到使用成员指针时，才提供成员所属的对象。

为了解释成员指针的原理，不妨使用 7.3.1 节（第 243 页）的 Screen 类：

836

```
class Screen {
public:
    typedef std::string::size_type pos;
    char get_cursor() const { return contents[cursor]; }
    char get() const;
```

```
    char get(pos ht, pos wd) const;
private:
    std::string contents;
    pos cursor;
    pos height, width;
};
```

19.4.1 数据成员指针

和其他指针一样，在声明成员指针时我们也使用*来表示当前声明的名字是一个指针。与普通指针不同的是，成员指针还必须包含成员所属的类。因此，我们必须在*之前添加*classname*::以表示当前定义的指针可以指向*classname*的成员。例如：

```
// pdata 可以指向一个常量（非常量）Screen 对象的 string 成员
const string Screen::*pdata;
```

上述语句将pdata声明成“一个指向Screen类的const string成员的指针”。常量对象的数据成员本身也是常量，因此将我们的指针声明成指向const string成员的指针意味着pdata可以指向任何Screen对象的一个成员，而不管该Screen对象是否是常量。作为交换条件，我们只能使用pdata读取它所指的成员，而不能向它写入内容。

当我们初始化一个成员指针（或者向它赋值）时，需指定它所指的成员。例如，我们可以令pdata指向某个非特定Screen对象的contents成员：

```
pdata = &Screen::contents;
```

其中，我们将取地址运算符作用于Screen类的成员而非内存中的一个该类对象。

当然，在C++11新标准中声明成员指针最简单的方法是使用auto或decltype：

```
auto pdata = &Screen::contents;
```

使用数据成员指针

读者必须清楚的一点是，当我们初始化一个成员指针或为成员指针赋值时，该指针并没有指向任何数据。成员指针指定了成员而非该成员所属的对象，只有当解引用成员指针时我们才提供对象的信息。

837

与成员访问运算符.和->类似，也有两种成员指针访问运算符：.*和->*，这两个运算符使得我们可以解引用指针并获得该对象的成员：

```
Screen myScreen, *pScreen = &myScreen;
// .*解引用 pdata 以获得 myScreen 对象的 contents 成员
auto s = myScreen.*pdata;
// ->*解引用 pdata 以获得 pScreen 所指对象的 contents 成员
s = pScreen->*pdata;
```

从概念上来说，这些运算符执行两步操作：它们首先解引用成员指针以得到所需的成员；然后像成员访问运算符一样，通过对象（.*）或指针（->*）获取成员。

返回数据成员指针的函数

常规的访问控制规则对成员指针同样有效。例如，Screen的contents成员是私有的，因此之前对于pdata的使用必须位于Screen类的成员或友元内部，否则程序将发生错误。

因为数据成员一般情况下是私有的，所以我们通常不能直接获得数据成员的指针。如果一个像 Screen 这样的类希望我们可以访问它的 contents 成员，最好定义一个函数，令其返回值是指向该成员的指针：

```
class Screen {
public:
    // data 是一个静态成员，返回一个成员指针
    static const std::string Screen::*data()
        { return &Screen::contents; }
    // 其他成员与之前的版本一致
};
```

我们为 Screen 类添加了一个静态成员，令其返回指向 contents 成员的指针。显然该函数的返回类型与最初的 pdata 指针类型一致。从右向左阅读函数的返回类型，可知 data 返回的是一个指向 Screen 类的 const string 成员的指针。函数体对 contents 成员使用了取地址运算符，因此函数将返回指向 Screen 类 contents 成员的指针。

当我们调用 data 函数时，将得到一个成员指针：

```
// data()返回一个指向 Screen 类的 contents 成员的指针
const string Screen::*pdata = Screen::data();
```

一如往常，pdata 指向 Screen 类的成员而非实际数据。要想使用 pdata，必须把它绑定到 Screen 类型的对象上：

```
// 获得 myScreen 对象的 contents 成员
auto s = myScreen.*pdata;
```

19.4.1 节练习

838

练习 19.11：普通的数据指针与指向数据成员的指针有何区别？

练习 19.12：定义一个成员指针，令其可以指向 Screen 类的 cursor 成员。通过该指针获得 Screen::cursor 的值。

练习 19.13：定义一个类型，使其可以表示指向 Sales_data 类的 bookNo 成员的指针。

19.4.2 成员函数指针

我们也可以定义指向类的成员函数的指针。与指向数据成员的指针类似，对于我们来说要想创建一个指向成员函数的指针，最简单的方法是使用 auto 来推断类型：

```
// pmf 是一个指针，它可以指向 Screen 的某个常量成员函数
// 前提是该函数不接受任何实参，并且返回一个 char
auto pmf = &Screen::get_cursor;
```

和指向数据成员的指针一样，我们使用 *classname*::*的形式声明一个指向成员函数的指针。类似于任何其他函数指针（参见 6.7 节，第 221 页），指向成员函数的指针也需要指定目标函数的返回类型和形参列表。如果成员函数是 const 成员（参见 7.1.2 节，第 231 页）或者引用成员（参见 13.6.3 节，第 483 页），则我们必须将 const 限定符或引用限定符包含进来。

和普通的函数指针类似，如果成员存在重载的问题，则我们必须显式地声明函数类型以明确指出我们想要使用的是哪个函数（参见 6.7 节，第 211 页）。例如，我们可以声明一

个指针，令其指向含有两个形参的 get：

```
char (Screen::*pmf2)(Screen::pos, Screen::pos) const;
pmf2 = &Screen::get;
```

出于优先级的考虑，上述声明中 Screen::*两端的括号必不可少。如果没有这对括号的话，编译器将认为该声明是一个（无效的）函数声明：

```
// 错误：非成员函数 p 不能使用 const 限定符
char Screen::*p(Screen::pos, Screen::pos) const;
```

这个声明试图定义一个名为 p 的普通函数，并且返回 Screen 类的一个 char 成员。因为它声明的是一个普通函数，所以不能使用 const 限定符。

和普通函数指针不同的是，在成员函数和指向该成员的指针之间不存在自动转换规则：

```
// pmf 指向一个 Screen 成员，该成员不接受任何实参且返回类型是 char
pmf = &Screen::get;        // 必须显式地使用取地址运算符
pmf = Screen::get;         // 错误：在成员函数和指针之间不存在自动转换规则
```

839 使用成员函数指针

和使用指向数据成员的指针一样，我们使用.*或者->*运算符作用于指向成员函数的指针，以调用类的成员函数：

```
Screen myScreen, *pScreen = &myScreen;
// 通过 pScreen 所指的对象调用 pmf 所指的函数
char c1 = (pScreen->*pmf)();
// 通过 myScreen 对象将实参 0, 0 传给含有两个形参的 get 函数
char c2 = (myScreen.*pmf2)(0, 0);
```

之所以(myScreen->*pmf)()和(pScreen.*pmf2)(0,0)的括号必不可少，原因是调用运算符的优先级要高于指针指向成员运算符的优先级。

假设去掉括号的话，

```
myScreen.*pmf()
```

其含义将等同于下面的式子：

```
myScreen.*(pmf())
```

这行代码的意思是调用一个名为 pmf 的函数，然后使用该函数的返回值作为指针指向成员运算符（.*）的运算对象。然而 pmf 并不是一个函数，因此代码将发生错误。



因为函数调用运算符的优先级较高，所以在声明指向成员函数的指针并使用这样的指针进行函数调用时，括号必不可少：(C::*p)(parms) 和(obj.*p)(args)。

使用成员指针的类型别名

使用类型别名或 typedef（参见 2.5.1 节，第 60 页）可以让成员指针更容易理解。例如，下面的类型别名将 Action 定义为两参数 get 函数的同义词：

```
// Action 是一种可以指向 Screen 成员函数的指针，它接受两个 pos 实参，返回一个 char
using Action =
char (Screen::*)(Screen::pos, Screen::pos) const;
```

Action 是某类型的另外一个名字，该类型是“指向 Screen 类的常量成员函数的指针，其中这个成员函数接受两个 pos 形参，返回一个 char”。通过使用 Action，我们可以简化指向 get 的指针定义：

```
Action get = &Screen::get;          // get 指向 Screen 的 get 成员
```

和其他函数指针类似，我们可以将指向成员函数的指针作为某个函数的返回类型或形参类型。其中，指向成员的指针形参也可以拥有默认实参：

```
// action 接受一个 Screen 的引用，和一个指向 Screen 成员函数的指针
Screen& action(Screen&, Action = &Screen::get);
```

action 是包含两个形参的函数，其中一个形参是 Screen 对象的引用，另一个形参是指向 Screen 成员函数的指针，成员函数必须接受两个 pos 形参并返回一个 char。当我们调用 action 时，只需将 Screen 的一个符合要求的函数的指针或地址传入即可： 840

```
Screen myScreen;
// 等价的调用：
action(myScreen);                   // 使用默认实参
action(myScreen, get);              // 使用我们之前定义的变量 get
action(myScreen, &Screen::get);     // 显式地传入地址
```



通过使用类型别名，可以令含有成员指针的代码更易读写。

成员指针函数表

对于普通函数指针和指向成员函数的指针来说，一种常见的用法是将其存入一个函数表当中（参见 14.8.3 节，第 511 页）。如果一个类含有几个相同类型的成员，则这样一张表可以帮助我们从这些成员中选择一个。假定 Screen 类含有几个成员函数，每个函数负责将光标向指定的方向移动：

```
class Screen {
public:
    // 其他接口和实现成员与之前一致
    Screen& home();                 // 光标移动函数
    Screen& forward();
    Screen& back();
    Screen& up();
    Screen& down();
};
```

这几个新函数有一个共同点：它们都不接受任何参数，并且返回值是发生光标移动的 Screen 的引用。

我们希望定义一个 move 函数，使其可以调用上面的任意一个函数并执行对应的操作。为了支持这个新函数，我们将在 Screen 中添加一个静态成员，该成员是指向光标移动函数的指针的数组：

```
class Screen {
public:
    // 其他接口和实现成员与之前一致
    // Action 是一个指针，可以用任意一个光标移动函数对其赋值
    using Action = Screen& (Screen::*)();
    // 指定具体要移动的方向，其中 enum 参见 19.3 节（第 736 页）
```

```
    enum Directions { HOME, FORWARD, BACK, UP, DOWN };
    Screen& move(Directions);
private:
    static Action Menu[];              // 函数表
};
```

841 数组 Menu 依次保存每个光标移动函数的指针，这些函数将按照 Directions 中枚举成员对应的偏移量存储。move 函数接受一个枚举成员并调用相应的函数：

```
Screen& Screen::move(Directions cm)
{
    // 运行 this 对象中索引值为 cm 的元素
    return (this->*Menu[cm])();   // Menu[cm]指向一个成员函数
}
```

move 中的函数调用的原理是：首先获取索引值为 cm 的 Menu 元素，该元素是指向 Screen 成员函数的指针。我们根据 this 所指的对象调用该元素所指的成员函数。

当我们调用 move 函数时，给它传入一个表示光标移动方向的枚举成员：

```
Screen myScreen;
myScreen.move(Screen::HOME); // 调用 myScreen.home
myScreen.move(Screen::DOWN); // 调用 myScreen.down
```

剩下的工作就是定义并初始化函数表本身了：

```
Screen::Action Screen::Menu[] = { &Screen::home,
                                  &Screen::forward,
                                  &Screen::back,
                                  &Screen::up,
                                  &Screen::down,
                                };
```

19.4.2 节练习

练习 19.14：下面的代码合法吗？如果合法，代码的含义是什么？如果不合法，解释原因。

```
auto pmf = &Screen::get_cursor;
pmf = &Screen::get;
```

练习 19.15：普通函数指针和指向成员函数的指针有何区别？

练习 19.16：声明一个类型别名，令其作为指向 Sales_data 的 avg_price 成员的指针的同义词。

练习 19.17：为 Screen 的所有成员函数类型各定义一个类型别名。

842 ## 19.4.3 将成员函数用作可调用对象

如我们所知，要想通过一个指向成员函数的指针进行函数调用，必须首先利用.*运算符或->*运算符将该指针绑定到特定的对象上。因此与普通的函数指针不同，成员指针不是一个可调用对象，这样的指针不支持函数调用运算符（参见 10.3.2 节，第 346 页）。

因为成员指针不是可调用对象，所以我们不能直接将一个指向成员函数的指针传递给算法。举个例子，如果我们想在一个 string 的 vector 中找到第一个空 string，显然

不能使用下面的语句：

```
auto fp = &string::empty; // fp 指向 string 的 empty 函数
// 错误，必须使用.*或->*调用成员指针
find_if(svec.begin(), svec.end(), fp);
```

find_if 算法需要一个可调用对象，但我们提供给它的是一个指向成员函数的指针 fp。因此在 find_if 的内部将执行如下形式的代码，从而导致无法通过编译：

```
// 检查对当前元素的断言是否为真
if (fp(*it))                    // 错误：要想通过成员指针调用函数，必须使用->*运算符
```

显然该语句试图调用的是传入的对象，而非函数。

使用 function 生成一个可调用对象

从指向成员函数的指针获取可调用对象的一种方法是使用标准库模板 function（参见 14.8.3 节，第 511 页）：

```
function<bool (const string&)> fcn = &string::empty;
find_if(svec.begin(), svec.end(), fcn);
```

我们告诉 function 一个事实：即 empty 是一个接受 string 参数并返回 bool 值的函数。通常情况下，执行成员函数的对象将被传给隐式的 this 形参。当我们想要使用 function 为成员函数生成一个可调用对象时，必须首先“翻译”该代码，使得隐式的形参变成显式的。

当一个 function 对象包含有一个指向成员函数的指针时，function 类知道它必须使用正确的指向成员的指针运算符来执行函数调用。也就是说，我们可以认为在 find_if 当中含有类似于如下形式的代码：

```
// 假设 it 是 find_if 内部的迭代器，则*it 是给定范围内的一个对象
if (fcn(*it))                   // 假设 fcn 是 find_if 内部的一个可调用对象的名字
```

其中，function 将使用正确的指向成员的指针运算符。从本质上来看，function 类将函数调用转换成了如下形式：

```
// 假设 it 是 find_if 内部的迭代器，则*it 是给定范围内的一个对象
if (((*it).*p)())               // 假设 p 是 fcn 内部的一个指向成员函数的指针
```

当我们定义一个 function 对象时，必须指定该对象所能表示的函数类型，即可调用对象的形式。如果可调用对象是一个成员函数，则第一个形参必须表示该成员是在哪个（一般是隐式的）对象上执行的。同时，我们提供给 function 的形式中还必须指明对象是否是以指针或引用的形式传入的。

以定义 fcn 为例，我们想在 string 对象的序列上调用 find_if，因此我们要求 function 生成一个接受 string 对象的可调用对象。又因为我们的 vector 保存的是 string 的指针，所以必须指定 function 接受指针：

```
vector<string*> pvec;
function<bool (const string*)> fp = &string::empty;
// fp 接受一个指向 string 的指针，然后使用->*调用 empty
find_if(pvec.begin(), pvec.end(), fp);
```

843

使用 mem_fn 生成一个可调用对象

通过上面的介绍可知，要想使用 function，我们必须提供成员的调用形式。我们也

C++ 11 可以采取另外一种方法，通过使用标准库功能 **mem_fn** 来让编译器负责推断成员的类型。和 function 一样，mem_fn 也定义在 functional 头文件中，并且可以从成员指针生成一个可调用对象；和 function 不同的是，mem_fn 可以根据成员指针的类型推断可调用对象的类型，而无须用户显式地指定：

```
find_if(svec.begin(), svec.end(), mem_fn(&string::empty));
```

我们使用 mem_fn(&string::empty)生成一个可调用对象，该对象接受一个 string 实参，返回一个 bool 值。

mem_fn 生成的可调用对象可以通过对象调用，也可以通过指针调用：

```
auto f = mem_fn(&string::empty); // f 接受一个 string 或者一个 string*
f(*svec.begin());          // 正确：传入一个 string 对象，f 使用.*调用 empty
f(&svec[0]);               // 正确：传入一个 string 的指针，f 使用->*调用 empty
```

实际上，我们可以认为 mem_fn 生成的可调用对象含有一对重载的函数调用运算符：一个接受 string*，另一个接受 string&。

使用 bind 生成一个可调用对象

出于完整性的考虑，我们还可以使用 bind（参见 10.3.4 节，第 354 页）从成员函数生成一个可调用对象：

```
// 选择范围中的每个 string，并将其 bind 到 empty 的第一个隐式实参上
auto it = find_if(svec.begin(), svec.end(),
                  bind(&string::empty, _1));
```

和 function 类似的地方是，当我们使用 bind 时，必须将函数中用于表示执行对象的隐式形参转换成显式的。和 mem_fn 类似的地方是，bind 生成的可调用对象的第一个实参既可以是 string 的指针，也可以是 string 的引用：

```
auto f = bind(&string::empty, _1);
f(*svec.begin());     // 正确：实参是一个 string，f 使用.*调用 empty
f(&svec[0]);          // 正确：实参是一个 string 的指针，f 使用->*调用 empty
```

19.4.3 节练习

练习 19.18：编写一个函数，使用 count_if 统计在给定的 vector 中有多少个空 string。

练习 19.19：编写一个函数，令其接受 vector<Sales_data>并查找平均价格高于某个值的第一个元素。

19.5 嵌套类

一个类可以定义在另一个类的内部，前者称为**嵌套类**（nested class）或**嵌套类型**（nested type）。嵌套类常用于定义作为实现部分的类，比如我们在文本查询示例中使用的 QueryResult 类（参见 12.3 节，第 430 页）。

844 嵌套类是一个独立的类，与外层类基本没什么关系。特别是，外层类的对象和嵌套类的对象是相互独立的。在嵌套类的对象中不包含任何外层类定义的成员；类似的，在外层类的对象中也不包含任何嵌套类定义的成员。

嵌套类的名字在外层类作用域中是可见的，在外层类作用域之外不可见。和其他嵌套的名字一样，嵌套类的名字不会和别的作用域中的同一个名字冲突。

嵌套类中成员的种类与非嵌套类是一样的。和其他类类似，嵌套类也使用访问限定符来控制外界对其成员的访问权限。外层类对嵌套类的成员没有特殊的访问权限，同样，嵌套类对外层类的成员也没有特殊的访问权限。

嵌套类在其外层类中定义了一个类型成员。和其他成员类似，该类型的访问权限由外层类决定。位于外层类 public 部分的嵌套类实际上定义了一种可以随处访问的类型；位于外层类 protected 部分的嵌套类定义的类型只能被外层类及其友元和派生类访问；位于外层类 private 部分的嵌套类定义的类型只能被外层类的成员和友元访问。

声明一个嵌套类

我们为 12.3.2 节（第 432 页）的 TextQuery 类定义了一个名为 QueryResult 的配套类，这两个类密切相关。QueryResult 类的主要作用是表示 TextQuery 对象上 query 操作的结果，显然将 QueryResult 用作其他目的没有任何意义。为了充分体现这种紧密的相关性，我们可以把 QueryResult 定义成 TextQuery 的成员。

```
class TextQuery {
public:
    class QueryResult;                    // 嵌套类稍后定义
    // 其他成员与 12.3.2 节（第 432 页）一致
};
```

我们只需对原来的 TextQuery 类做一处改动，即将 QueryResult 声明成嵌套类。因为 QueryResult 是一个类型成员（参见 7.4.1 节，第 254 页），所以我们必须对它先声明后使用，尤其是必须先声明 QueryResult，再将它作为 query 成员的返回类型。类的其他成员没有任何变化。

在外层类之外定义一个嵌套类

845

我们在 TextQuery 内声明了 QueryResult，但是没有给出它的定义。和成员函数一样，嵌套类必须声明在类的内部，但是可以定义在类的内部或者外部。

当我们在外层类之外定义一个嵌套类时，必须以外层类的名字限定嵌套类的名字：

```
// QueryResult 是 TextQuery 的成员，下面的代码负责定义 QueryResult
class TextQuery::QueryResult {
    // 位于类的作用域内，因此我们不必对 QueryResult 形参进行限定
    friend std::ostream&
           print(std::ostream&, const QueryResult&);
public:
    // 无须定义 QueryResult::line_no
    // 嵌套类可以直接使用外层类的成员，无须对该成员的名字进行限定
    QueryResult(std::string,
                std::shared_ptr<std::set<line_no>>,
                std::shared_ptr<std::vector<std::string>>);
    // 其他成员与 12.3.2 节（第 432 页）一致
};
```

和原来的类相比唯一的改动是，我们无须在 QueryResult 内定义 line_no 成员了。因为该成员属于 TextQuery，所以 QueryResult 可以直接访问它而不必再定义一次。



在嵌套类在其外层类之外完成真正的定义之前，它都是一个不完全类型（参见 7.3.3 节，第 250 页）。

定义嵌套类的成员

在这个版本的 QueryResult 类中，我们并没有在类的内部定义其构造函数。要想为其定义构造函数，必须指明 QueryResult 是嵌套在 TextQuery 的作用域之内的。具体做法是使用外层类的名字限定嵌套类的名字：

```
// QueryResult 类嵌套在 TextQuery 类中
// 下面的代码为 QueryResult 类定义名为 QueryResult 的成员
TextQuery::QueryResult::QueryResult(string s,
                shared_ptr<set<line_no>> p,
                shared_ptr<vector<string>> f):
        sought(s), lines(p), file(f) { }
```

从右向左阅读函数的名字可知我们定义的是 QueryResult 类的构造函数，而 QueryResult 类是嵌套在 TextQuery 类中的。该构造函数除了把实参值赋给对应的数据成员之外，没有做其他工作。

嵌套类的静态成员定义

如果 QueryResult 声明了一个静态成员，则该成员的定义将位于 TextQuery 的作
846 用域之外。例如，假设 QueryResult 有一个静态成员，则该成员的定义将形如：

```
// QueryResult 类嵌套在 TextQuery 类中，
// 下面的代码为 QueryResult 定义一个静态成员
int TextQuery::QueryResult::static_mem = 1024;
```

嵌套类作用域中的名字查找

名字查找的一般规则（参见 7.4.1 节，第 254 页）在嵌套类中同样适用。当然，因为嵌套类本身是一个嵌套作用域，所以还必须查找嵌套类的外层作用域。这种作用域嵌套的性质正好可以说明为什么我们不在 QueryResult 的嵌套版本中定义 line_no。原来的 QueryResult 类定义了该成员，从而使其成员可以避免使用 TextQuery::line_no 的形式。然而 QueryResult 的嵌套类版本本身就是定义在 TextQuery 中的，所以我们不需要再使用 typedef。嵌套的 QueryResult 无须说明 line_no 属于 TextQuery 就可以直接使用它。

如我们所知，嵌套类是其外层类的一个类型成员，因此外层类的成员可以像使用任何其他类型成员一样使用嵌套类的名字。因为 QueryResult 嵌套在 TextQuery 中，所以 TextQuery 的 query 成员可以直接使用名字 QueryResult：

```
// 返回类型必须指明 QueryResult 是一个嵌套类
TextQuery::QueryResult
TextQuery::query(const string &sought) const
{
    // 如果我们没有找到 sought，则返回 set 的指针
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    // 使用 find 而非下标以避免向 wm 中添加单词
    auto loc = wm.find(sought);
    if (loc == wm.end())
```

```
            return QueryResult(sought, nodata, file);          // 没有找到
        else
            return QueryResult(sought, loc->second, file);
    }
```

和过去一样，返回类型不在类的作用域中（参见 7.4 节，第 253 页），因此我们必须指明函数的返回值是 TextQuery::QueryResult 类型。不过在函数体内部我们可以直接访问 QueryResult，比如上面的 return 语句就是这样。

嵌套类和外层类是相互独立的

尽管嵌套类定义在其外层类的作用域中，但是读者必须谨记外层类的对象和嵌套类的对象没有任何关系。嵌套类的对象只包含嵌套类定义的成员；同样，外层类的对象只包含外层类定义的成员，在外层类对象中不会有任何嵌套类的成员。

说得再具体一些，TextQuery::query 的第二条 return 语句

```
return QueryResult(sought, loc->second, file);
```

使用了 TextQuery 对象的数据成员，而 query 正是用它们来初始化 QueryResult 对象的。因为在一个 QueryResult 对象中不包含其外层类的成员，所以我们必须使用上述成员构造我们返回的 QueryResult 对象。 847

19.5 节练习

练习 19.20：将你的 QueryResult 类嵌套在 TextQuery 中，然后重新运行 12.3.2 节（第 435 页）中使用了 TextQuery 的程序。

19.6 union：一种节省空间的类

联合（union）是一种特殊的类。一个 union 可以有多个数据成员，但是在任意时刻只有一个数据成员可以有值。当我们给 union 的某个成员赋值之后，该 union 的其他成员就变成未定义的状态了。分配给一个 union 对象的存储空间至少要能容纳它的最大的数据成员。和其他类一样，一个 union 定义了一种新类型。

类的某些特性对 union 同样适用，但并非所有特性都如此。union 不能含有引用类型的成员，除此之外，它的成员可以是绝大多数类型。在 C++11 新标准中，含有构造函数或析构函数的类类型也可以作为 union 的成员类型。union 可以为其成员指定 public、protected 和 private 等保护标记。默认情况下，union 的成员都是公有的，这一点与 struct 相同。

union 可以定义包括构造函数和析构函数在内的成员函数。但是由于 union 既不能继承自其他类，也不能作为基类使用，所以在 union 中不能含有虚函数。

定义 union

union 提供了一种有效的途径使得我们可以方便地表示一组类型不同的互斥值。举个例子，假设我们需要处理一些不同种类的数字数据和字符数据，则在此过程中可以定义一个 union 来保存这些值：

```
// Token 类型的对象只有一个成员，该成员的类型可能是下列类型中的任意一种
union Token {
```

```
// 默认情况下成员是公有的
    char   cval;
    int    ival;
    double dval;
};
```

在定义一个 union 时，首先是关键字 union，随后是该 union 的（可选的）名字以及花括号内的一组成员声明。上面的代码定义了一个名为 Token 的 union，它可以保存一个值，这个值的类型可能是 char、int 或 double 中的一种。

848 使用 union 类型

union 的名字是一个类型名。和其他内置类型一样，默认情况下 union 是未初始化的。我们可以像显式地初始化聚合类（参见 7.5.5 节，第 266 页）一样使用一对花括号内的初始值显式地初始化一个 union：

```
Token first_token = {'a'};          // 初始化 cval 成员
Token last_token;                   // 未初始化的 Token 对象
Token *pt = new Token;              // 指向一个未初始化的 Token 对象的指针
```

如果提供了初始值，则该初始值被用于初始化第一个成员。因此，first_token 的初始化过程实际上是给 cval 成员赋了一个初值。

我们使用通用的成员访问运算符访问一个 union 对象的成员：

```
last_token.cval = 'z';
pt->ival = 42;
```

为 union 的一个数据成员赋值会令其他数据成员变成未定义的状态。因此，当我们使用 union 时，必须清楚地知道当前存储在 union 中的值到底是什么类型。如果我们使用错误的数据成员或者为错误的数据成员赋值，则程序可能崩溃或出现异常行为，具体的情况根据成员的类型而有所不同。

匿名 union

匿名 union（anonymous union）是一个未命名的 union，并且在右花括号和分号之间没有任何声明（参见 2.6.1 节，第 65 页）。一旦我们定义了一个匿名 union，编译器就自动地为该 union 创建一个未命名的对象：

```
union {                         // 匿名 union
    char cval;
    int ival;
    double dval;
}; // 定义一个未命名的对象，我们可以直接访问它的成员
cval = 'c';                     // 为刚刚定义的未命名的匿名 union 对象赋一个新值
ival = 42;                      // 该对象当前保存的值是 42
```

在匿名 union 的定义所在的作用域内该 union 的成员都是可以直接访问的。



匿名 union 不能包含受保护的成员或私有成员，也不能定义成员函数。

含有类类型成员的 union

C++的早期版本规定，在 union 中不能含有定义了构造函数或拷贝控制成员的类类型

成员。C++11 新标准取消了这一限制。不过，如果 union 的成员类型定义了自己的构造函数和/或拷贝控制成员，则该union的用法要比只含有内置类型成员的union复杂得多。 849 C++ 11

当 union 包含的是内置类型的成员时，我们可以使用普通的赋值语句改变 union 保存的值。但是对于含有特殊类类型成员的 union 就没这么简单了。如果我们想将 union 的值改为类类型成员对应的值，或者将类类型成员的值改为一个其他值，则必须分别构造或析构该类类型的成员：当我们将 union 的值改为类类型成员对应的值时，必须运行该类型的构造函数；反之，当我们将类类型成员的值改为一个其他值时，必须运行该类型的析构函数。

当 union 包含的是内置类型的成员时，编译器将按照成员的次序依次合成默认构造函数或拷贝控制成员。但是如果 union 含有类类型的成员，并且该类型自定义了默认构造函数或拷贝控制成员，则编译器将为 union 合成对应的版本并将其声明为删除的（参见 13.1.6 节，第 450 页）。

例如，string 类定义了五个拷贝控制成员以及一个默认构造函数。如果 union 含有 string 类型的成员，并且没有自定义默认构造函数或某个拷贝控制成员，则编译器将合成缺少的成员并将其声明成删除的。如果在某个类中含有一个 union 成员，而且该 union 含有删除的拷贝控制成员，则该类与之对应的拷贝控制操作也将是删除的。

使用类管理 union 成员

对于 union 来说，要想构造或销毁类类型的成员必须执行非常复杂的操作，因此我们通常把含有类类型成员的 union 内嵌在另一个类当中。这个类可以管理并控制与 union 的类类型成员有关的状态转换。举个例子，我们为 union 添加一个 string 成员，并将我们的 union 定义成匿名 union，最后将它作为 Token 类的一个成员。此时，Token 类将可以管理 union 的成员。

为了追踪 union 中到底存储了什么类型的值，我们通常会定义一个独立的对象，该对象称为 union 的**判别式**（discriminant）。我们可以使用判别式辨认 union 存储的值。为了保持 union 与其判别式同步，我们将判别式也作为 Token 的成员。我们的类将定义一个枚举类型（参见 19.3 节，第 736 页）的成员来追踪其 union 成员的状态。

在我们的类中定义的函数包括默认构造函数、拷贝控制成员以及一组赋值运算符，这些赋值运算符可以将 union 的某种类型的值赋给 union 成员：

```
class Token {
public:
    // 因为 union 含有一个 string 成员，所以 Token 必须定义拷贝控制成员
    // 定义移动构造函数和移动赋值运算符的任务留待本节练习完成
    Token(): tok(INT), ival{0} { }
    Token(const Token &t): tok(t.tok) { copyUnion(t); }
    Token &operator=(const Token&);
    // 如果 union 含有一个 string 成员，则我们必须销毁它，参见 19.1.2 节（第 729 页）
    ~Token() { if (tok == STR) sval.~string(); }
    // 下面的赋值运算符负责设置 union 的不同成员
    Token &operator=(const std::string&);
    Token &operator=(char);
    Token &operator=(int);
    Token &operator=(double);
private:
```

850

```
    enum {INT, CHAR, DBL, STR} tok; // 判别式
    union { // 匿名 union
        char   cval;
        int    ival;
        double dval;
        std::string sval;
    }; // 每个 Token 对象含有一个该未命名 union 类型的未命名成员
    // 检查判别式，然后酌情拷贝 union 成员
    void copyUnion(const Token&);
};
```

我们的类定义了一个嵌套的、未命名的、不限定作用域的枚举类型（参见 19.3 节，第 736 页），并将其作为 tok 成员的类型。其中，tok 的声明位于枚举类型定义的右侧花括号之后，以及表示该枚举类型定义结束的分号之前，因此，tok 的类型就是当前这个未命名的 enum 类型（参见 2.6.1 节，第 65 页）。

我们使用 tok 作为判别式。当 union 存储的是一个 int 值时，tok 的值是 INT；当 union 存储的是一个 string 值时，tok 的值是 STR；以此类推。

类的默认构造函数初始化判别式以及 union 成员，令其保存 int 值 0。

因为我们的 union 含有一个定义了析构函数的成员，所以必须为 union 也定义一个析构函数以销毁 string 成员。和普通的类类型成员不一样，作为 union 组成部分的类成员无法自动销毁。因为析构函数不清楚 union 存储的值是什么类型，所以它无法确定应该销毁哪个成员。

我们的析构函数检查被销毁的对象中是否存储着 string 值。如果有，则类的析构函数显式地调用 string 的析构函数（参见 19.1.2 节，第 729 页）释放该 string 使用的内存；反之，如果 union 存储的值是内置类型，则类的析构函数什么也不做。

管理判别式并销毁 string

类的赋值运算符将负责设置 tok 并为 union 的相应成员赋值。和析构函数一样，这些运算符在为 union 赋新值前必须首先销毁 string：

```
Token &Token::operator=(int i)
{
    if (tok == STR) sval.~string();   // 如果当前存储的是 string，释放它
    ival = i;                         // 为成员赋值
    tok = INT;                        // 更新判别式
    return *this;
}
```

如果 union 的当前值是 string，则我们必须先调用 string 的析构函数销毁这个 string，然后再为 union 赋新值。清除了 string 成员之后，我们将给定的值赋给与运算符形参类型相匹配的成员。在此例中，形参类型是 int，所以我们赋值给 ival。随后更新判别式并返回结果。

851

double 和 char 版本的赋值运算符与 int 赋值运算符非常相似，读者可以在本节的练习中尝试使用这两个运算符。string 版本与其他几个有所区别，原因是 string 版本必须管理与 string 类型有关的转换：

```
Token &Token::operator=(const std::string &s)
{
```

```
    if (tok == STR)                    // 如果当前存储的是 string，可以直接赋值
        sval = s;
    else
        new(&sval) string(s);          // 否则需要先构造一个 string
    tok = STR;                         // 更新判别式
    return *this;
}
```

在此例中，如果 union 当前存储的是 string，则我们可以使用普通的 string 赋值运算符直接为其赋值。如果 union 当前存储的不是 string，则我们找不到一个已存在的 string 对象供我们调用赋值运算符。此时，我们必须先利用定位 new 表达式（参见 19.1.2 节，第 729 页）在内存中为 sval 构造一个 string，然后将该 string 初始化为 string 形参的副本，最后更新判别式并返回结果。

管理需要拷贝控制的联合成员

和依赖于类型的赋值运算符一样，拷贝构造函数和赋值运算符也需要先检验判别式以明确拷贝所采用的方式。为了完成这一任务，我们定义一个名为 copyUnion 的成员。

当我们在拷贝构造函数中调用 copyUnion 时，union 成员将被默认初始化，这意味着编译器会初始化 union 的第一个成员。因为 string 不是第一个成员，所以显然 union 成员保存的不是 string。在赋值运算符中情况有些不一样，有可能 union 已经存储了一个 string。我们将在赋值运算符中直接处理这种情况。copyUnion 假设如果它的形参存储了 string，则它一定会构造自己的 string：

```
void Token::copyUnion(const Token &t)
{
    switch (t.tok) {
        case Token::INT: ival = t.ival; break;
        case Token::CHAR: cval = t.cval; break;
        case Token::DBL: dval = t.dval; break;
        // 要想拷贝一个 string 可以使用定位 new 表达式构造它，参见 19.1.2 节（第 729 页）
        case Token::STR: new(&sval) string(t.sval); break;
    }
}
```

该函数使用一个 switch 语句（参见 5.3.2 节，第 159 页）检验判别式。对于内置类型来说，我们把值直接赋给对应的成员；如果拷贝的是一个 string，则需要构造它。 852

赋值运算符必须处理 string 成员的三种可能情况：左侧运算对象和右侧运算对象都是 string、两个运算对象都不是 string、只有一个运算对象是 string：

```
Token &Token::operator=(const Token &t)
{
    // 如果此对象的值是 string 而 t 的值不是，则我们必须释放原来的 string
    if (tok == STR && t.tok != STR) sval.~string();
    if (tok == STR && t.tok == STR)
        sval = t.sval; // 无须构造一个新 string
    else
        copyUnion(t);  // 如果 t.tok 是 STR，则需要构造一个 string
    tok = t.tok;
    return *this;
}
```

如果作为左侧运算对象的 union 的值是 string 但右侧运算对象的值不是，则我们必须先释放原来的 string 再给 union 成员赋一新值。如果两侧运算对象的值都是 string，则我们可以使用普通的 string 赋值运算符完成拷贝。否则，我们调用 copyUnion 进行赋值。在 copyUnion 内部，如果右侧运算对象是 string，则我们在左侧运算对象的 union 成员内构造一个新 string；如果两端都不是 string，则直接执行普通的赋值操作就可以了。

19.6 节练习

练习 19.21：编写你自己的 Token 类。

练习 19.22：为你的 Token 类添加一个 Sales_data 类型的成员。

练习 19.23：为你的 Token 类添加移动构造函数和移动赋值运算符。

练习 19.24：如果我们将一个 Token 对象赋给它自己将发生什么情况？

练习 19.25：编写一系列赋值运算符，令其分别接受 union 中各种类型的值。

19.7 局部类

类可以定义在某个函数的内部，我们称这样的类为**局部类**（local class）。局部类定义的类型只在定义它的作用域内可见。和嵌套类不同，局部类的成员受到严格限制。



局部类的所有成员（包括函数在内）都必须完整定义在类的内部。因此，局部类的作用与嵌套类相比相差很远。

853

在实际编程的过程中，因为局部类的成员必须完整定义在类的内部，所以成员函数的复杂性不可能太高。局部类的成员函数一般只有几行代码，否则我们就很难读懂它了。

类似的，在局部类中也不允许声明静态数据成员，因为我们没法定义这样的成员。

局部类不能使用函数作用域中的变量

局部类对其外层作用域中名字的访问权限受到很多限制，局部类只能访问外层作用域定义的类型名、静态变量（参见 6.1.1 节，第 185 页）以及枚举成员。如果局部类定义在某个函数内部，则该函数的普通局部变量不能被该局部类使用：

```
int a, val;
void foo(int val)
{
    static int si;
    enum Loc { a = 1024, b };
    // Bar 是 foo 的局部类
    struct Bar {
        Loc locVal;                    // 正确：使用一个局部类型名
        int barVal;

        void fooBar(Loc l = a)         // 正确：默认实参是 Loc::a
        {
            barVal = val;              // 错误：val 是 foo 的局部变量
```

```
            barVal = ::val;          // 正确：使用一个全局对象
            barVal = si;             // 正确：使用一个静态局部对象
            locVal = b;              // 正确：使用一个枚举成员
        }
    };
    // ...
}
```

常规的访问保护规则对局部类同样适用

外层函数对局部类的私有成员没有任何访问特权。当然，局部类可以将外层函数声明为友元；或者更常见的情况是局部类将其成员声明成公有的。在程序中有权访问局部类的代码非常有限。局部类已经封装在函数作用域中，通过信息隐藏进一步封装就显得没什么必要了。

局部类中的名字查找

局部类内部的名字查找次序与其他类相似。在声明类的成员时，必须先确保用到的名字位于作用域中，然后再使用该名字。定义成员时用到的名字可以出现在类的任意位置。如果某个名字不是局部类的成员，则继续在外层函数作用域中查找；如果还没有找到，则在外层函数所在的作用域中查找。

854

嵌套的局部类

可以在局部类的内部再嵌套一个类。此时，嵌套类的定义可以出现在局部类之外。不过，嵌套类必须定义在与局部类相同的作用域中。

```
void foo()
{
    class Bar {
    public:
        // ...
        class Nested; // 声明 Nested 类
    };
    // 定义 Nested 类
    class Bar::Nested {
        // ...
    };
}
```

和往常一样，当我们在类的外部定义成员时，必须指明该成员所属的作用域。因此在上面的例子中，`Bar::Nested` 的意思是 `Nested` 是定义在 `Bar` 的作用域内的一个类。

局部类内的嵌套类也是一个局部类，必须遵循局部类的各种规定。嵌套类的所有成员都必须定义在嵌套类内部。

19.8 固有的不可移植的特性

为了支持低层编程，C++定义了一些固有的**不可移植**（nonportable）的特性。所谓不可移植的特性是指因机器而异的特性，当我们将含有不可移植特性的程序从一台机器转移到另一台机器上时，通常需要重新编写该程序。算术类型的大小在不同机器上不一样（参见 2.1.1 节，第 30 页），这是我们使用过的不可移植特性的一个典型示例。

本节将介绍 C++从 C 语言继承而来的另外两种不可移植的特性：位域和 volatile 限定符。此外，我们还将介绍链接指示，它是 C++新增的一种不可移植的特性。

19.8.1 位域

类可以将其（非静态）数据成员定义成**位域**（bit-field），在一个位域中含有一定数量的二进制位。当一个程序需要向其他程序或硬件设备传递二进制数据时，通常会用到位域。



位域在内存中的布局是与机器相关的。

855

位域的类型必须是整型或枚举类型（参见 19.3 节，第 736 页）。因为带符号位域的行为是由具体实现确定的，所以在通常情况下我们使用无符号类型保存一个位域。位域的声明形式是在成员名字之后紧跟一个冒号以及一个常量表达式，该表达式用于指定成员所占的二进制位数：

```
typedef unsigned int Bit;
class File {
    Bit mode: 2;                    // mode 占 2 位
    Bit modified: 1;                // modified 占 1 位
    Bit prot_owner: 3;              // prot_owner 占 3 位
    Bit prot_group: 3;              // prot_group 占 3 位
    Bit prot_world: 3;              // prot_world 占 3 位
    // File 的操作和数据成员
public:
    // 文件类型以八进制的形式表示，参见 2.1.3 节（第 35 页）
    enum modes { READ = 01, WRITE = 02, EXECUTE = 03 };
    File &open(modes);
    void close();
    void write();
    bool isRead() const;
    void setWrite();
};
```

mode 位域占 2 个二进制位，modified 只占 1 个，其他成员则各占 3 个。如果可能的话，在类的内部连续定义的位域压缩在同一整数的相邻位，从而提供存储压缩。例如在之前的声明中，五个位域可能会存储在同一个 unsigned int 中。这些二进制位是否能压缩到一个整数中以及如何压缩是与机器相关的。

取地址运算符（&）不能作用于位域，因此任何指针都无法指向类的位域。



通常情况下最好将位域设为无符号类型，存储在带符号类型中的位域的行为将因具体实现而定。

使用位域

访问位域的方式与访问类的其他数据成员的方式非常相似：

```
void File::write()
{
    modified = 1;
```

```
    // ...
}
void File::close()
{
    if (modified)
        // …… 保存内容
}
```

通常使用内置的位运算符（参见 4.8 节，第 136 页）操作超过 1 位的位域：

856

```
File &File::open(File::modes m)
{
    mode |= READ;              // 按默认方式设置 READ
    // 其他处理
    if (m & WRITE)             // 如果打开了 READ 和 WRITE
    // 按照读/写方式打开文件
    return *this;
}
```

如果一个类定义了位域成员，则它通常也会定义一组内联的成员函数以检验或设置位域的值：

```
inline bool File::isRead() const { return mode & READ; }
inline void File::setWrite() { mode |= WRITE; }
```

19.8.2 volatile 限定符



volatile 的确切含义与机器有关，只能通过阅读编译器文档来理解。要想让使用了 volatile 的程序在移植到新机器或新编译器后仍然有效，通常需要对该程序进行某些改变。

直接处理硬件的程序常常包含这样的数据元素，它们的值由程序直接控制之外的过程控制。例如，程序可能包含一个由系统时钟定时更新的变量。当对象的值可能在程序的控制或检测之外被改变时，应该将该对象声明为 **volatile**。关键字 volatile 告诉编译器不应对这样的对象进行优化。

volatile 限定符的用法和 const 很相似，它起到对类型额外修饰的作用：

```
volatile int display_register;    // 该 int 值可能发生改变
volatile Task *curr_task;         // curr_task 指向一个 volatile 对象
volatile int iax[max_size];       // iax 的每个元素都是 volatile
volatile Screen bitmapBuf;        // bitmapBuf 的每个成员都是 volatile
```

const 和 volatile 限定符互相没什么影响，某种类型可能既是 const 的也是 volatile 的，此时它同时具有二者的属性。

就像一个类可以定义 const 成员函数一样，它也可以将成员函数定义成 volatile 的。只有 volatile 的成员函数才能被 volatile 的对象调用。

2.4.2 节（第 56 页）描述了 const 限定符和指针的相互作用，在 volatile 限定符和指针之间也存在类似的关系。我们可以声明 volatile 指针、指向 volatile 对象的指针以及指向 volatile 对象的 volatile 指针：

```
volatile int v;        // v 是一个 volatile int
```

857

```
int *volatile vip;    // vip是一个volatile指针，它指向int
volatile int *ivp;    // ivp是一个指针，它指向一个volatile int
// vivp是一个volatile指针，它指向一个volatile int
volatile int *volatile vivp;

int *ip = &v;         // 错误：必须使用指向volatile的指针
ivp = &v;             // 正确：ivp是一个指向volatile的指针
vivp = &v;            // 正确：vivp是一个指向volatile的volatile指针
```

和 const 一样，我们只能将一个 volatile 对象的地址（或者拷贝一个指向 volatile 类型的指针）赋给一个指向 volatile 的指针。同时，只有当某个引用是 volatile 的时，我们才能使用一个 volatile 对象初始化该引用。

合成的拷贝对 volatile 对象无效

const 和 volatile 的一个重要区别是我们不能使用合成的拷贝/移动构造函数及赋值运算符初始化 volatile 对象或从 volatile 对象赋值。合成的成员接受的形参类型是（非 volatile）常量引用，显然我们不能把一个非 volatile 引用绑定到一个 volatile 对象上。

如果一个类希望拷贝、移动或赋值它的 volatile 对象，则该类必须自定义拷贝或移动操作。例如，我们可以将形参类型指定为 const volatile 引用，这样我们就能利用任意类型的 Foo 进行拷贝或赋值操作了：

```
class Foo {
public:
    Foo(const volatile Foo&); // 从一个volatile对象进行拷贝
    // 将一个volatile对象赋值给一个非volatile对象
    Foo& operator=(volatile const Foo&);
    // 将一个volatile对象赋值给一个volatile对象
    Foo& operator=(volatile const Foo&) volatile;
    // Foo类的剩余部分
};
```

尽管我们可以为 volatile 对象定义拷贝和赋值操作，但是一个更深层次的问题是拷贝 volatile 对象是否有意义呢？不同程序使用 volatile 的目的各不相同，对上述问题的回答与具体的使用目的密切相关。

19.8.3 链接指示：extern "C"

C++程序有时需要调用其他语言编写的函数，最常见的是调用 C 语言编写的函数。像所有其他名字一样，其他语言中的函数名字也必须在 C++中进行声明，并且该声明必须指定返回类型和形参列表。对于其他语言编写的函数来说，编译器检查其调用的方式与处理
858
普通 C++函数的方式相同，但是生成的代码有所区别。C++使用**链接指示**（linkage directive）指出任意非 C++函数所用的语言。



要想把 C++代码和其他语言（包括 C 语言）编写的代码放在一起使用，要求我们必须有权访问该语言的编译器，并且这个编译器与当前的 C++编译器是兼容的。

声明一个非 C++的函数

链接指示可以有两种形式：单个的或复合的。链接指示不能出现在类定义或函数定义的内部。同样的链接指示必须在函数的每个声明中都出现。

举个例子，接下来的声明显示了 cstring 头文件的某些 C 函数是如何声明的：

```
// 可能出现在 C++头文件<cstring>中的链接指示
// 单语句链接指示
extern "C" size_t strlen(const char *);
// 复合语句链接指示
extern "C" {
    int strcmp(const char*, const char*);
    char *strcat(char*, const char*);
}
```

链接指示的第一种形式包含一个关键字 extern，后面是一个字符串字面值常量以及一个“普通的”函数声明。

其中的字符串字面值常量指出了编写函数所用的语言。编译器应该支持对 C 语言的链接指示。此外，编译器也可能会支持其他语言的链接指示，如 extern "Ada"、 extern "FORTRAN"等。

链接指示与头文件

我们可以令链接指示后面跟上花括号括起来的若干函数的声明，从而一次性建立多个链接。花括号的作用是将适用于该链接指示的多个声明聚合在一起，否则花括号就会被忽略，花括号中声明的函数名字就是可见的，就好像在花括号之外声明的一样。

多重声明的形式可以应用于整个头文件。例如，C++的 cstring 头文件可能形如：

```
// 复合语句链接指示
extern "C" {
#include <string.h>        // 操作 C 风格字符串的 C 函数
}
```

当一个#include 指示被放置在复合链接指示的花括号中时，头文件中的所有普通函数声明都被认为是由链接指示的语言编写的。链接指示可以嵌套，因此如果头文件包含带自带链接指示的函数，则该函数的链接不受影响。 859



C++从 C 语言继承的标准库函数可以定义成 C 函数，但并非必须：决定使用 C 还是 C++实现 C 标准库，是每个 C++实现的事情。

指向 extern "C"函数的指针

编写函数所用的语言是函数类型的一部分。因此，对于使用链接指示定义的函数来说，它的每个声明都必须使用相同的链接指示。而且，指向其他语言编写的函数的指针必须与函数本身使用相同的链接指示：

```
// pf 指向一个 C 函数，该函数接受一个 int 返回 void
extern "C" void (*pf)(int);
```

当我们使用 pf 调用函数时，编译器认定当前调用的是一个 C 函数。

指向 C 函数的指针与指向 C++函数的指针是不一样的类型。一个指向 C 函数的指针

不能用在执行初始化或赋值操作后指向 C++函数，反之亦然。就像其他类型不匹配的问题一样，如果我们试图在两个链接指示不同的指针之间进行赋值操作，则程序将发生错误：

```
void (*pf1)(int);                    // 指向一个 C++函数
extern "C" void (*pf2)(int);         // 指向一个 C 函数
pf1 = pf2;                           // 错误：pf1 和 pf2 的类型不同
```



有的 C++编译器会接受之前的这种赋值操作并将其作为对语言的扩展，尽管从严格意义上来看它是非法的。

链接指示对整个声明都有效

当我们使用链接指示时，它不仅对函数有效，而且对作为返回类型或形参类型的函数指针也有效：

```
// f1 是一个 C 函数，它的形参是一个指向 C 函数的指针
extern "C" void f1(void(*)(int));
```

这条声明语句指出 f1 是一个不返回任何值的 C 函数。它有一个类型是函数指针的形参，其中的函数接受一个 int 形参返回为空。这个链接指示不仅对 f1 有效，对函数指针同样有效。当我们调用 f1 时，必须传给它一个 C 函数的名字或者指向 C 函数的指针。

因为链接指示同时作用于声明语句中的所有函数，所以如果我们希望给 C++函数传入一个指向 C 函数的指针，则必须使用类型别名（参见 2.5.1 节，第 60 页）：

860

```
// FC 是一个指向 C 函数的指针
extern "C" typedef void FC(int);
// f2 是一个 C++函数，该函数的形参是指向 C 函数的指针
void f2(FC *);
```

导出 C++函数到其他语言

通过使用链接指示对函数进行定义，我们可以令一个 C++函数在其他语言编写的程序中可用：

```
// calc 函数可以被 C 程序调用
extern "C" double calc(double dparm) { /* ... */ }
```

编译器将为该函数生成适合于指定语言的代码。

值得注意的是，可被多种语言共享的函数的返回类型或形参类型受到很多限制。例如，我们不太可能把一个 C++类的对象传给 C 程序，因为 C 程序根本无法理解构造函数、析构函数以及其他类特有的操作。

对链接到 C 的预处理器的支持

有时需要在 C 和 C++中编译同一个源文件，为了实现这一目的，在编译 C++版本的程序时预处理器定义__cplusplus（两个下画线）。利用这个变量，我们可以在编译 C++程序的时候有条件地包含进来一些代码：

```
#ifdef __cplusplus
// 正确：我们正在编译 C++程序
extern "C"
#endif
int strcmp(const char*, const char*);
```

重载函数与链接指示

链接指示与重载函数的相互作用依赖于目标语言。如果目标语言支持重载函数，则为该语言实现链接指示的编译器很可能也支持重载这些C++的函数。

C语言不支持函数重载，因此也就不难理解为什么一个C链接指示只能用于说明一组重载函数中的某一个了：

```
// 错误：两个 extern "C"函数的名字相同
extern "C" void print(const char*);
extern "C" void print(int);
```

如果在一组重载函数中有一个是C函数，则其余的必定都是C++函数：

861

```
class SmallInt { /* ... */ };
class BigNum { /* ... */ };
// C函数可以在C或C++程序中调用
// C++函数重载了该函数，可以在C++程序中调用
extern "C" double calc(double);
extern SmallInt calc(const SmallInt&);
extern BigNum calc(const BigNum&);
```

C版本的calc函数可以在C或C++程序中调用，而使用了类类型形参的C++函数只能在C++程序中调用。上述性质与声明的顺序无关。

19.8.3 节练习

练习 19.26：说明下列声明语句的含义并判断它们是否合法：

```
extern "C" int compute(int *, int);
extern "C" double compute(double *, double);
```

862 小结

C++为解决某些特殊问题设置了一系列特殊的处理机制。

有的程序需要精确控制内存分配过程，它们可以通过在类的内部或在全局作用域中自定义 operator new 和 operator delete 来实现这一目的。如果应用程序为这两个操作定义了自己的版本，则 new 和 delete 表达式将优先使用应用程序定义的版本。

有的程序需要在运行时直接获取对象的动态类型，运行时类型识别（RTTI）为这种程序提供了语言级别的支持。RTTI 只对定义了虚函数的类有效；对没有定义虚函数的类，虽然也可以得到其类型信息，但只是静态类型。

当我们定义指向类成员的指针时，在指针类型中包含了该指针所指成员所属类的类型信息。成员指针可以绑定到该类当中任意一个具有指定类型的成员上。当我们解引用成员指针时，必须提供获取成员所需的对象。

C++定义了另外几种聚集类型：

- 嵌套类，定义在其他类的作用域中，嵌套类通常作为外层类的实现类。
- union，是一种特殊的类，它可以定义几个数据成员但是在任意时刻只有一个成员有值，union 通常嵌套在其他类的内部。
- 局部类，定义在函数的内部，局部类的所有成员都必须定义在类内，局部类不能含有静态数据成员。

C++支持几种固有的不可移植的特性，其中位域和 volatile 使得程序更容易访问硬件；链接指示使得程序更容易访问用其他语言编写的代码。

术语表

匿名 union（anonymous union） 未命名的 union，不能用于定义对象。匿名 union 的成员也是外层作用域的成员。匿名 union 不能包含成员函数，也不能包含私有成员或受保护的成员。

位域（bit-field） 特殊的类成员，该成员含有一个整型值以指定为其分配的二进制位数。如果可能的话，在类中连续定义的位域将被压缩在一个普通的整数值当中。

判别式（discriminant） 是一种使用一个对象判断 union 的当前值类型的编程技术。

dynamic_cast 是一个运算符，执行从基类向派生类的带检查的强制类型转换。当基类中至少含有一个虚函数时，该运算符负
863 责检查指针或引用所绑定的对象的动态类型。如果对象类型与目标类型（或其派生类）一致，则类型转换完成。否则，指针转换将返回一个值为 0 的指针；引用转换将抛出一个异常。

枚举类型（enumeration） 将一组整型常量命名后聚合在一起形成的类型。

枚举成员（enumerator） 是枚举类型的成员。枚举成员是常量，可以用在任何需要整型常量的地方。

free 是定义在 cstdlib 中的低层函数，负责释放内存。free 只能释放由 malloc 分配的内存。

链接指示（linkage directive） 支持 C++程序调用其他语言编写的函数的一种机制。所有编译器都应支持调用 C++和 C 函数，至于是否支持其他语言则由编译器决定。

局部类（local class） 定义在函数中的类。局部类只有在其外层函数内可见。局部类

的所有成员都必须定义在类的内部。局部类不能含有静态成员。局部类成员不能访问外层函数的非静态变量，只能访问类型名字、静态变量或枚举成员。

malloc 是定义在cstdlib中的低层函数，负责分配内存。malloc 分配的内存必须由 free 释放。

mem_fn 是一个标准库类模板，根据指向成员函数的指针生成一个可调用对象。

嵌套类（nested class） 定义在其他类内部的类，嵌套类定义在它的外层作用域中：在外层类的作用域中嵌套类的名字必须唯一，在外层类之外可以被重用。在外层类之外访问嵌套类需要用作用域运算符指明嵌套类所属的范围。

嵌套类型（nested type） “嵌套类”的同义词。

不可移植（nonportable） 固有的与机器有关的特性，当程序转移到其他机器或编译器上时需要修改代码。

operator delete 是一个标准库函数，用于释放由 operator new 分配的未指明类型的、未构造的内存空间。相应的，operator delete[]释放由 operator new[]为数组分配的内存。

operator new 是一个标准库函数，用于分配一个给定大小的、未指明类型的、未构造的内存空间。标准库函数 operator new[] 为数组分配原始内存。与 allocator 类相比，这两个标准库函数提供的内存分配机制更低级。现代的 C++程序应该使用 allocator 而不是这两个函数。

定位 new 表达式（placement new expression） 是 new 的一种特殊形式，在给定的内存中构造对象。它不分配内存，而是根据实参指定在哪儿构造对象。它是对 allocator 类的 construct 成员的行为的一种低级模拟。

成员指针（pointer to member） 其中既包含类类型，也包含指针所指的成员类型。成员指针的定义必须同时指定类的名字以及指针所指的成员类型：

```
T C::*pmem = &C::member;
```

该语句将 pmem 定义为一个指针，它可以指向类 C 的成员，并且该成员的类型是 T，然后初始化 pmem 令其指向类 C 的名为 member 的成员。要使用该指针，我们必须提供 C 的一个对象或指针：

```
classobj.*pmem;

classptr->*pmem;
```

从 classptr 所指的对象 classobj 中获取 member。 864

运行时类型识别（run-time type identification） 是 C++的一种特性，允许在运行时获取指针或引用的动态类型。RTTI 运算符包括 typeid 和 dynamic_cast，为含有虚函数的类的指针或引用提供动态类型。当作用于其他类型时，返回的结果是指针或引用的静态类型。

限定作用域的枚举类型（scoped enumeration） 是一种新的枚举类型，它的枚举成员不能被外层作用域直接访问。

typeid 运算符（typeid operator） 是一个一元运算符，返回标准库类型 type_info 的引用，表示给定表达式的类型。当表达式是某个含有虚函数的类型的对象时，返回表达式的动态类型；此类表达式在运行时求值。如果表达式的类型是指针、引用或其他未定义虚函数的类型，则返回指针、引用或对象的静态类型；此类表达式不会被求值。

type_info typeid 运算符返回的标准库类型。type_info 的细节因机器而异，但是必须提供一组操作，其中名为 name 的函数负责返回一个表示类型名字的字符串。type_info 对象不能被拷贝、移动或赋值。

联合（union） 是一种和类有些相似的类型。可以包含多个数据成员，但是同一时刻只能有一个成员有值。联合可以有包括

构造函数和析构函数在内的成员函数。联合不能被用作基类。在 C++11 新标准中，联合可以含有类类型的成员，前提是这些类自定义了拷贝控制成员。对于这样的联合来说，如果它们没有定义自己的拷贝控制成员，则编译器将为它们生成删除的版本。

不限定作用域的枚举类型（unscoped enumeration） 该枚举类型的枚举成员在枚举类型的外层作用域中可以访问。

volatile 是一种类型限定符，告诉编译器变量可能在程序的直接控制之外发生改变。它起到一种标示的作用，令编译器不对代码进行优化操作。

附录 A
标准库

内容

A.1 标准库名字和头文件 766
A.2 算法概览 770
A.3 随机数 781

本附录介绍了标准库中算法和随机数部分的一些额外细节。这里还提供了一个我们使用过的所有标准库名字的列表，列表中给出了每个名字所在的头文件。

在第 10 章中我们使用过一些较常用的算法，并且描述了算法之下的架构。在本附录中，我们将列出所有标准库算法，按它们执行的操作的种类来组织。

在 17.4 节（第 660 页）中我们描述了随机数库的架构，并使用了几个分布类型。库中定义了若干随机数引擎和 20 种不同的分布。在本附录中，我们将列出所有引擎和分布类型。

866

A.1 标准库名字和头文件

本书中大多数代码没有给出编译程序所需的实际#include 指令。为了方便读者，表 A.1 列出了本书程序用到的标准库名字以及它们所在的头文件。

表 A.1：标准库名字和头文件

名字	头文件
abort	<cstdlib>
accumulate	<numeric>
allocator	<memory>
array	<array>
auto_ptr	<memory>
back_inserter	<iterator>
bad_alloc	<new>
bad_array_new_length	<new>
bad_cast	<typeinfo>
begin	<iterator>
bernoulli_distribution	<random>
bind	<functional>
bitset	<bitset>
boolalpha	<iostream>
cerr	<iostream>
cin	<iostream>
cmatch	<regex>
copy	<algorithm>
count	<algorithm>
count_if	<algorithm>
cout	<iostream>
cref	<functional>
csub_match	<regex>
dec	<iostream>
default_float_engine	<iostream>
default_random_engine	<random>
deque	<deque>
domain_error	<stdexcept>
end	<iterator>
endl	<iostream>
ends	<iostream>
equal_range	<algorithm>
exception	<exception>
fill	<algorithm>
fill_n	<algorithm>
find	<algorithm>

续表

名字	头文件
find_end	<algorithm>
find_first_of	<algorithm>
find_if	<algorithm>
fixed	<iostream>
flush	<iostream>
for_each	<algorithm>
forward	<utility>
forward_list	<forward_list>
free	<cstdlib>
front_inserter	<iterator>
fstream	<fstream>
function	<functional>
get	<tuple>
getline	<string>
greater	<functional>
hash	<functional>
hex	<iostream>
hexfloat	<iostream>
ifstream	<fstream>
initializer_list	<initializer_list>
inserter	<iterator>
internal	<iostream>
ios_base	<ios_base>
isalpha	<cctype>
islower	<cctype>
isprint	<cctype>
ispunct	<cctype>
isspace	<cctype>
istream	<iostream>
istream_iterator	<iterator>
istringstream	<sstream>
isupper	<cctype>
left	<iostream>
less	<functional>
less_equal	<functional>
list	<list>
logic_error	<stdexcept>
lower_bound	<algorithm>
lround	<cmath>
make_move_iterator	<iterator>
make_pair	<utility>

867

续表

名字	头文件
make_shared	<memory>
make_tuple	<tuple>
malloc	<cstdlib>
map	<map>
868 max	<algorithm>
max_element	<algorithm>
mem_fn	<functional>
min	<algorithm>
move	<utility>
multimap	<map>
multiset	<set>
negate	<functional>
noboolalpha	<iostream>
normal_distribution	<random>
noshowbase	<iostream>
noshowpoint	<iostream>
noskipws	<iostream>
not1	<functional>
nothrow	<new>
nothrow_t	<new>
nounitbuf	<iostream>
nouppercase	<iostream>
nth_element	<algorithm>
oct	<iostream>
ofstream	<fstream>
ostream	<iostream>
ostream_iterator	<iterator>
ostringstream	<sstream>
out_of_range	<stdexcept>
pair	<utility>
partial_sort	<algorithm>
placeholders	<functional>
placeholders::_1	<functional>
plus	<functional>
priority_queue	<queue>
ptrdiff_t	<cstddef>
queue	<queue>
rand	<random>
random_device	<random>
range_error	<stdexcept>
ref	<functional>

续表

名字	头文件
regex	<regex>
regex_constants	<regex>
regex_error	<regex>
regex_match	<regex>
regex_replace	<regex>
regex_search	<regex>
remove_pointer	<type_traits>
remove_reference	<type_traits>
replace	<algorithm>
replace_copy	<algorithm>
reverse_iterator	<iterator>
right	<iostream>
runtime_error	<stdexcept>
scientific	<iostream>
set	<set>
set_difference	<algorithm>
set_intersection	<algorithm>
set_union	<algorithm>
setfill	<iomanip>
setprecision	<iomanip>
setw	<iomanip>
shared_ptr	<memory>
showbase	<iostream>
showpoint	<iostream>
size_t	<cstddef>
skipws	<iostream>
smatch	<regex>
sort	<algorithm>
sqrt	<cmath>
sregex_iterator	<regex>
ssub_match	<regex>
stable_sort	<algorithm>
stack	<stack>
stoi	<string>
strcmp	<cstring>
strcpy	<cstring>
string	<string>
stringstream	<sstream>
strlen	<cstring>
strncpy	<cstring>
strtod	<string>

869

续表

名字	头文件
swap	<utility>
terminate	<exception>
time	<ctime>
tolower	<cctype>
toupper	<cctype>
transform	<algorithm>
tuple	<tuple>
tuple_element	<tuple>
tuple_size	<tuple>
type_info	<typeinfo>
unexpected	<exception>
uniform_int_distribution	<random>
uniform_real_distribution	<random>
uninitialized_copy	<memory>
uninitialized_fill	<memory>
unique	<algorithm>
unique_copy	<algorithm>
unique_ptr	<memory>
unitbuf	<iostream>
unordered_map	<unordered_map>
unordered_multimap	<unordered_map>
unordered_multiset	<unordered_set>
unordered_set	<unordered_set>
upper_bound	<algorithm>
uppercase	<iostream>
vector	<vector>
weak_ptr	<memory>

870

A.2 算法概览

标准库定义了超过 100 个算法。要想高效使用这些算法需要了解它们的结构而不是单纯记忆每个算法的细节。因此，我们在第 10 章中关注标准库算法架构的描述和理解。在本节中，我们将简要描述每个算法，在下面的描述中，

- beg 和 end 是表示元素范围的迭代器（参见 9.2.1 节，第 296 页）。几乎所有算法都对一个由 beg 和 end 表示的序列进行操作。
- beg2 是表示第二个输入序列开始位置的迭代器。end2 表示第二个序列的末尾位置（如果有的话）。如果没有 end2，则假定 beg2 表示的序列与 beg 和 end 表示的序列一样大。beg 和 beg2 的类型不必匹配，但是，必须保证对两个序列中的元素都可以执行特定操作或调用给定的可调用对象。
- dest 是表示目的序列的迭代器。对于给定输入序列，算法需要生成多少元素，目的序列必须保证能保存同样多的元素。

- unaryPred 和 binaryPred 是一元和二元谓词（参见 10.3.1 节，第 344 页），分别接受一个和两个参数，都是来自输入序列的元素，两个谓词都返回可用作条件的类型。
- comp 是一个二元谓词，满足关联容器中对关键字序的要求（参见 11.2.2 节，第 378 页）。 871
- unaryOp 和 binaryOp 是可调用对象（参见 10.3.2 节，第 346 页），可分别使用来自输入序列的一个和两个实参来调用。

A.2.1 查找对象的算法

这些算法在一个输入序列中搜索一个指定值或一个值的序列。

每个算法都提供两个重载的版本，第一个版本使用底层类型的相等运算符（==）来比较元素；第二个版本使用用户给定的 unaryPred 和 binaryPred 比较元素。

简单查找算法

这些算法查找指定值，要求输入迭代器（input iterator）。

```
find(beg, end, val)
find_if(beg, end, unaryPred)
find_if_not(beg, end, unaryPred)
count(beg, end, val)
count_if(beg, end, unaryPred)
```

find 返回一个迭代器，指向输入序列中第一个等于 val 的元素。

find_if 返回一个迭代器，指向第一个满足 unaryPred 的元素。

find_if_not 返回一个迭代器，指向第一个令 unaryPred 为 false 的元素。上述三个算法在未找到元素时都返回 end。

count 返回一个计数器，指出 val 出现了多少次；count_if 统计有多少个元素满足 unaryPred。

```
all_of(beg, end, unaryPred)
any_of(beg, end, unaryPred)
none_of(beg, end, unaryPred)
```

这些算法都返回一个 bool 值，分别指出 unaryPred 是否对所有元素都成功、对任意一个元素成功以及对所有元素都不成功。如果序列为空，any_of 返回 false，而 all_of 和 none_of 返回 true。

查找重复值的算法

下面这些算法要求前向迭代器（forward iterator），在输入序列中查找重复元素。

```
adjacent_find(beg, end)
adjacent_find(beg, end, binaryPred)
```

返回指向第一对相邻重复元素的迭代器。如果序列中无相邻重复元素，则返回 end。

```
search_n(beg, end, count, val)
search_n(beg, end, count, val, binaryPred)
```

返回一个迭代器，从此位置开始有 count 个相等元素。如果序列中不存在这样的子序列，

则返回 end。

872 **查找子序列的算法**

在下面的算法中，除了 find_first_of 之外，都要求两个前向迭代器。find_first_of 用输入迭代器表示第一个序列，用前向迭代器表示第二个序列。这些算法搜索子序列而不是单个元素。

```
search(beg1, end1, beg2, end2)
search(beg1, end1, beg2, end2, binaryPred)
```

返回第二个输入范围（子序列）在第一个输入范围中第一次出现的位置。如果未找到子序列，则返回 end1。

```
find_first_of(beg1, end1, beg2, end2)
find_first_of(beg1, end1, beg2, end2, binaryPred)
```

返回一个迭代器，指向第二个输入范围中任意元素在第一个范围中首次出现的位置。如果未找到匹配元素，则返回 end1。

```
find_end(beg1, end1, beg2, end2)
find_end(beg1, end1, beg2, end2, binaryPred)
```

类似 search，但返回的是最后一次出现的位置。如果第二个输入范围为空，或者在第一个输入范围中未找到它，则返回 end1。

A.2.2 其他只读算法

这些算法要求前两个实参都是输入迭代器。

equal 和 mismatch 算法还接受一个额外的输入迭代器，表示第二个范围的开始位置。这两个算法都提供两个重载的版本。第一个版本使用底层类型的相等运算符（==）比较元素，第二个版本则用用户指定的 unaryPred 或 binaryPred 比较元素。

```
for_each(beg, end, unaryOp)
```

对输入序列中的每个元素应用可调用对象（参见 10.3.2 节，第 346 页）unaryOp。unaryOp 的返回值（如果有的话）被忽略。如果迭代器允许通过解引用运算符向序列中的元素写入值，则 unaryOp 可能修改元素。

```
mismatch(beg1, end1, beg2)
mismatch(beg1, end1, beg2, binaryPred)
```

比较两个序列中的元素。返回一个迭代器的 pair（参见 11.2.3 节，第 379 页），表示两个序列中第一个不匹配的元素。如果所有元素都匹配，则返回的 pair 中第一个迭代器为 end1，第二个迭代器指向 beg2 中偏移量等于第一个序列长度的位置。

```
equal(beg1, end1, beg2)
equal(beg1, end1, beg2, binaryPred)
```

确定两个序列是否相等。如果输入序列中每个元素都与从 beg2 开始的序列中对应元素相等，则返回 true。

873 ## A.2.3 二分搜索算法

这些算法都要求前向迭代器，但这些算法都经过了优化，如果我们提供随机访问迭代

器（random-access iterator）的话，它们的性能会好得多。从技术上讲，无论我们提供什么类型的迭代器，这些算法都会执行对数次的比较操作。但是，当使用前向迭代器时，这些算法必须花费线性次数的迭代器操作来移动到序列中要比较的元素。

这些算法要求序列中的元素已经是有序的。它们的行为类似关联容器的同名成员（参见 11.3.5 节，第 389 页）。equal_range、lower_bound 和 upper_bound 算法返回迭代器，指向给定元素在序列中的正确插入位置——插入后还能保持有序。如果给定元素比序列中的所有元素都大，则会返回尾后迭代器。

每个算法都提供两个版本：第一个版本用元素类型的小于运算符（<）来检测元素；第二个版本则使用给定的比较操作。在下列算法中，“x 小于 y”表示 x<y 或 comp(x,y) 成功。

```
lower_bound(beg, end, val)
lower_bound(beg, end, val, comp)
```

返回一个迭代器，表示第一个小于等于 val 的元素，如果不存在这样的元素，则返回 end。

```
upper_bound(beg, end, val)
upper_bound(beg, end, val, comp)
```

返回一个迭代器，表示第一个大于 val 的元素，如果不存在这样的元素，则返回 end。

```
equal_range(beg, end, val)
equal_range(beg, end, val, comp)
```

返回一个 pair（参见 11.2.3 节，第 379 页），其 first 成员是 lower_bound 返回的迭代器，second 成员是 upper_bound 返回的迭代器。

```
binary_search(beg, end, val)
binary_search(beg, end, val, comp)
```

返回一个 bool 值，指出序列中是否包含等于 val 的元素。对于两个值 x 和 y，当 x 不小于 y 且 y 也不小于 x 时，认为它们相等。

A.2.4 写容器元素的算法

很多算法向给定序列中的元素写入新值。这些算法可以从不同角度加以区分：通过表示输入序列的迭代器类型来区分；或者通过是写入输入序列中元素还是写入给定目的位置来区分。

只写不读元素的算法

874

这些算法要求一个输出迭代器（output iterator），表示目的位置。_n 结尾的版本接受第二个实参，表示写入的元素数目，并将给定数目的元素写入到目的位置中。

```
fill(beg, end, val)
fill_n(dest, cnt, val)
generate(beg, end, Gen)
generate_n(dest, cnt, Gen)
```

给输入序列中每个元素赋予一个新值。fill 将值 val 赋予元素；generate 执行生成器对象 Gen() 生成新值。生成器是一个可调用对象（参见 10.3.2 节，第 346 页），每次调用会生成一个不同的返回值。fill 和 generate 都返回 void。_n 版本返回一个迭代器，指向写入到输出序列的最后一个元素之后的位置。

使用输入迭代器的写算法

这些算法读取一个输入序列，将值写入到一个输出序列中。它们要求一个名为 dest 的输出迭代器，而表示输入范围的迭代器必须是输入迭代器。

```
copy(beg, end, dest)
copy_if(beg, end, dest, unaryPred)
copy_n(beg, n, dest)
```

从输入范围将元素拷贝到 dest 指定的目的序列。copy 拷贝所有元素，copy_if 拷贝那些满足 unaryPred 的元素，copy_n 拷贝前 n 个元素。输入序列必须有至少 n 个元素。

```
move(beg, end, dest)
```

对输入序列中的每个元素调用 std:: move（参见 13.6.1 节，第 472 页），将其移动到迭代器 dest 开始的序列中。

```
transform(beg, end, dest, unaryOp)
transform(beg, end, beg2, dest, binaryOp)
```

调用给定操作，并将结果写到 dest 中。第一个版本对输入范围中每个元素应用一元操作。第二个版本对两个输入序列中的元素应用二元操作。

```
replace_copy(beg, end, dest, old_val, new_val)
replace_copy_if(beg, end, dest, unaryPred, new_val)
```

将每个元素拷贝到 dest，将指定的元素替换为 new_val。第一个版本替换那些==old_val 的元素。第二个版本替换那些满足 unaryPred 的元素。

```
merge(beg1, end1, beg2, end2, dest)
merge(beg1, end1, beg2, end2, dest, comp)
```

两个输入序列必须都是有序的。将合并后的序列写入到 dest 中。第一个版本用<运算符比较元素；第二个版本则使用给定比较操作。

875

使用前向迭代器的写算法

这些算法要求前向迭代器，由于它们是向输入序列写入元素，迭代器必须具有写入元素的权限。

```
iter_swap(iter1, iter2)
swap_ranges(beg1, end1, beg2)
```

交换 iter1 和 iter2 所表示的元素，或将输入范围中所有元素与 beg2 开始的第二个序列中所有元素进行交换。两个范围不能有重叠。iter_swap 返回 void，swap_ranges 返回递增后的 beg2，指向最后一个交换元素之后的位置。

```
replace(beg, end, old_val, new_val)
replace_if(beg, end, unaryPred, new_val)
```

用 new_val 替换每个匹配元素。第一个版本使用==比较元素与 old_val，第二个版本替换那些满足 unaryPred 的元素。

使用双向迭代器的写算法

这些算法需要在序列中有反向移动的能力，因此它们要求双向迭代器（bidirectional iterator）。

```
copy_backward(beg, end, dest)
```