TextQuery 1 文本查询程序 1

C++ Primer 5th Edition, Chapter 12

12.2.2 节练习

练习 12.26：用 allocator 重写第 427 页中的程序。

12.3 使用标准库：文本查询程序

我们将实现一个简单的文本查询程序，作为标准库相关内容学习的总结。我们的程序允许用户在一个给定文件中查询单词。查询结果是单词在文件中出现的次数及其所在行的列表。如果一个单词在一行中出现多次，此行只列出一次。行会按照升序输出——即，第 7 行会在第 9 行之前显示，依此类推。

例如，我们可能读入一个包含本章内容（指英文版中的文本）的文件，在其中寻找单词 element。输出结果的前几行应该是这样的：

```
element occurs 112 times
    (line 36) A set element contains only a key;
    (line 158) operator creates a new element
    (line 160) Regardless of whether the element
    (line 168) When we fetch an element from a map, we
    (line 214) If the element is not found, find returns
```

接下来还有大约 100 行，都是单词 element 出现的位置。

12.3.1 文本查询程序设计

485

开始一个程序的设计的一种好方法是列出程序的操作。了解需要哪些操作会帮助我们分析出需要什么样的数据结构。从需求入手，我们的文本查询程序需要完成如下任务：

- 当程序读取输入文件时，它必须记住单词出现的每一行。因此，程序需要逐行读取输入文件，并将每一行分解为独立的单词
- 当程序生成输出时，
  — 它必须能提取每个单词所关联的行号
  — 行号必须按升序出现且无重复
  — 它必须能打印给定行号中的文本。

利用多种标准库设施，我们可以很漂亮地实现这些要求：

- 我们将使用一个 vector<string>来保存整个输入文件的一份拷贝。输入文件中的每行保存为 vector 中的一个元素。当需要打印一行时，可以用行号作为下标来提取行文本。
- 我们使用一个 istringstream（参见 8.3 节，第 287 页）来将每行分解为单词。
- 我们使用一个 set 来保存每个单词在输入文本中出现的行号。这保证了每行只出现一次且行号按升序保存。
- 我们使用一个 map 来将每个单词与它出现的行号 set 关联起来。这样我们就可以方便地提取任意单词的 set。

我们的解决方案还使用了 shared_ptr，原因稍后进行解释。

数据结构

虽然我们可以用 vector、set 和 map 来直接编写文本查询程序，但如果定义一个更

为抽象的解决方案，会更为有效。我们将从定义一个保存输入文件的类开始，这会令文件查询更为容易。我们将这个类命名为 TextQuery，它包含一个 vector 和一个 map。vector 用来保存输入文件的文本，map 用来关联每个单词和它出现的行号的 set。这个类将会有一个用来读取给定输入文件的构造函数和一个执行查询的操作。

查询操作要完成的任务非常简单：查找 map 成员，检查给定单词是否出现。设计这个函数的难点是确定应该返回什么内容。一旦找到了一个单词，我们需要知道它出现了多少次、它出现的行号以及每行的文本。

返回所有这些内容的最简单的方法是定义另一个类，可以命名为 QueryResult，来保存查询结果。这个类会有一个 print 函数，完成结果打印工作。

在类之间共享数据

486

我们的 QueryResult 类要表达查询的结果。这些结果包括与给定单词关联的行号的 set 和这些行对应的文本。这些数据都保存在 TextQuery 类型的对象中。

由于 QueryResult 所需要的数据都保存在一个 TextQuery 对象中，我们就必须确定如何访问它们。我们可以拷贝行号的 set，但这样做可能很耗时。而且，我们当然不希望拷贝 vector，因为这可能会引起整个文件的拷贝，而目标只不过是为了打印文件的一小部分而已（通常会是这样）。

通过返回指向 TextQuery 对象内部的迭代器（或指针），我们可以避免拷贝操作。但是，这种方法开启了一个陷阱：如果 TextQuery 对象在对应的 QueryResult 对象之前被销毁，会发生什么？在此情况下，QueryResult 就将引用一个不再存在的对象中的数据。

对于 QueryResult 对象和对应的 TextQuery 对象的生存期应该同步这一观察结果，其实已经暗示了问题的解决方案。考虑到这两个类概念上“共享”了数据，可以使用 shared_ptr（参见 12.1.1 节，第 400 页）来反映数据结构中的这种共享关系。

使用 TextQuery 类

当我们设计一个类时，在真正实现成员之前先编写程序使用这个类，是一种非常有用的方法。通过这种方法，可以看到类是否具有我们所需要的操作。例如，下面的程序使用了 TextQuery 和 QueryResult 类。这个函数接受一个指向要处理的文件的 ifstream，并与用户交互，打印给定单词的查询结果

```
void runQueries(ifstream &infile)
{
    // infile 是一个 ifstream，指向我们要处理的文件
    TextQuery tq(infile); // 保存文件并建立查询 map
    // 与用户交互：提示用户输入要查询的单词，完成查询并打印结果
    while (true) {
        cout << "enter word to look for, or q to quit: ";
        string s;
        // 若遇到文件尾或用户输入了'q'时循环终止
        if (!(cin >> s) || s == "q") break;
        // 指向查询并打印结果
        print(cout, tq.query(s)) << endl;
    }
}
```

我们首先用给定的 ifstream 初始化一个名为 tq 的 TextQuery 对象。TextQuery 的构造函数读取输入文件，保存在 vector 中，并建立单词到所在行号的 map。

while（无限）循环提示用户输入一个要查询的单词，并打印出查询结果，如此往复。循环条件检测字面常量 true（参见 2.1.3 节，第 37 页），因此永远成功。循环的退出是通
487 过 if 语句中的 break（参见 5.5.1 节，第 170 页）实现的。此 if 语句检查输入是否成功。如果成功，它再检查用户是否输入了 q。输入失败或用户输入了 q 都会使循环终止。一旦用户输入了要查询的单词，我们要求 tq 查找这个单词，然后调用 print 打印搜索结果。

12.3.1 节练习

练习 12.27：TextQuery 和 QueryResult 类只使用了我们已经介绍过的语言和标准库特性。不要提前看后续章节内容，只用已经学到的知识对这两个类编写你自己的版本。

练习 12.28：编写程序实现文本查询，不要定义类来管理数据。你的程序应该接受一个文件，并与用户交互来查询单词。使用 vector、map 和 set 容器来保存来自文件的数据并生成查询结果。

练习 12.29：我们曾经用 do while 循环来编写管理用户交互的循环（参见 5.4.4 节，第 169 页）。用 do while 重写本节程序，解释你倾向于哪个版本，为什么。

12.3.2 文本查询程序类的定义

我们以 TextQuery 类的定义开始。用户创建此类的对象时会提供一个 istream，用来读取输入文件。这个类还提供一个 query 操作，接受一个 string，返回一个 QueryResult 表示 string 出现的那些行。

设计类的数据成员时，需要考虑与 QueryResult 对象共享数据的需求。QueryResult 类需要共享保存输入文件的 vector 和保存单词关联的行号的 set。因此，这个类应该有两个数据成员：一个指向动态分配的 vector（保存输入文件）的 shared_ptr 和一个 string 到 shared_ptr<set>的 map。map 将文件中每个单词关联到一个动态分配的 set 上，而此 set 保存了该单词所出现的行号。

为了使代码更易读，我们还会定义一个类型成员（参见 7.3.1 节，第 243 页）来引用行号，即 string 的 vector 中的下标：

```
class QueryResult; // 为了定义函数 query 的返回类型，这个定义是必需的
class TextQuery {
public:
    using line_no = std::vector<std::string>::size_type;
    TextQuery(std::ifstream&);
    QueryResult query(const std::string&) const;
private:
    std::shared_ptr<std::vector<std::string>> file; // 输入文件
    // 每个单词到它所在的行号的集合的映射
    std::map<std::string,
             std::shared_ptr<std::set<line_no>>> wm;
};
```

488 这个类定义最困难的部分是解开类名。与往常一样，对于可能置于头文件中的代码，在使用标准库名字时要加上 std::（参见 3.1 节，第 74 页）。在本例中，我们反复使用了 std::，

使得代码开始可能有些难读。例如，

```
std::map<std::string, std::shared_ptr<std::set<line_no>>> wm;
```

如果写成下面的形式可能就更好理解一些

```
map<string, shared_ptr<set<line_no>>> wm;
```

TextQuery 构造函数

TextQuery 的构造函数接受一个 ifstream，逐行读取输入文件：

```
// 读取输入文件并建立单词到行号的映射
TextQuery::TextQuery(ifstream &is): file(new vector<string>)
{
    string text;
    while (getline(is, text)) {          // 对文件中每一行
        file->push_back(text);           // 保存此行文本
        int n = file->size() - 1;        // 当前行号
        istringstream line(text);        // 将行文本分解为单词
        string word;
        while (line >> word) {           // 对行中每个单词
            // 如果单词不在 wm 中，以之为下标在 wm 中添加一项
            auto &lines = wm[word];  // lines 是一个 shared_ptr
            if (!lines) // 在我们第一次遇到这个单词时，此指针为空
                lines.reset(new set<line_no>); // 分配一个新的 set
            lines->insert(n);            // 将此行号插入 set 中
        }
    }
}
```

构造函数的初始化器分配一个新的 vector 来保存输入文件中的文本。我们用 getline 逐行读取输入文件，并存入 vector 中。由于 file 是一个 shared_ptr，我们用->运算符解引用 file 来提取 file 指向的 vector 对象的 push_back 成员。

接下来我们用一个 istringstream（参见 8.3 节，第 287 页）来处理刚刚读入的一行中的每个单词。内层 while 循环用 istringstream 的输入运算符来从当前行读取每个单词，存入 word 中。在 while 循环内，我们用 map 下标运算符提取与 word 相关联的 shared_ptr<set>，并将 lines 绑定到此指针。注意，lines 是一个引用，因此改变 lines 也会改变 wm 中的元素。

若 word 不在 map 中，下标运算符会将 word 添加到 wm 中（参见 11.3.4 节，第 387 页），与 word 关联的值进行值初始化。这意味着，如果下标运算符将 word 添加到 wm 中，lines 将是一个空指针。如果 lines 为空，我们分配一个新的 set，并调用 reset 更新 lines 引用的 shared_ptr，使其指向这个新分配的 set。

不管是否创建了一个新的 set，我们都调用 insert 将当前行号添加到 set 中。由于 lines 是一个引用，对 insert 的调用会将新元素添加到 wm 中的 set 中。如果一个给定单词在同一行中出现多次，对 insert 的调用什么都不会做。 489

QueryResult 类

QueryResult 类有三个数据成员：一个 string，保存查询单词；一个 shared_ptr，指向保存输入文件的 vector；一个 shared_ptr，指向保存单词出现行号的 set。它唯

一的一个成员函数是一个构造函数，初始化这三个数据成员：

```
class QueryResult {
friend std::ostream& print(std::ostream&, const QueryResult&);
public:
QueryResult(std::string s,
            std::shared_ptr<std::set<line_no>> p,
            std::shared_ptr<std::vector<std::string>> f):
    sought(s), lines(p), file(f) { }
private:
    std::string sought; // 查询单词
    std::shared_ptr<std::set<line_no>> lines;          // 出现的行号
    std::shared_ptr<std::vector<std::string>> file;    // 输入文件
};
```

构造函数的唯一工作是将参数保存在对应的数据成员中，这是在其初始化器列表中完成的（参见 7.1.4 节，第 237 页）。

query 函数

query 函数接受一个 string 参数，即查询单词，query 用它来在 map 中定位对应的行号 set。如果找到了这个 string，query 函数构造一个 QueryResult，保存给定 string、TextQuery 的 file 成员以及从 wm 中提取的 set。

唯一的问题是：如果给定 string 未找到，我们应该返回什么？在这种情况下，没有可返回的 set。为了解决此问题，我们定义了一个局部 static 对象，它是一个指向空的行号 set 的 shared_ptr。当未找到给定单词时，我们返回此对象的一个拷贝：

```
QueryResult
TextQuery::query(const string &sought) const
{
    // 如果未找到 sought，我们将返回一个指向此 set 的指针
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    // 使用 find 而不是下标运算符来查找单词，避免将单词添加到 wm 中！
    auto loc = wm.find(sought);
    if (loc == wm.end())
        return QueryResult(sought, nodata, file); // 未找到
    else
        return QueryResult(sought, loc->second, file);
}
```

490 ### 打印结果

print 函数在给定的流上打印出给定的 QueryResult 对象：

```
ostream &print(ostream & os, const QueryResult &qr)
{
    // 如果找到了单词，打印出现次数和所有出现的位置
    os << qr.sought << " occurs " << qr.lines->size() << " "
       << make_plural(qr.lines->size(), "time", "s") << endl;
    // 打印单词出现的每一行
    for (auto num : *qr.lines) // 对 set 中每个单词
        // 避免行号从 0 开始给用户带来的困惑
        os << "\t(line " << num + 1 << ") "
```

```
            << *(qr.file->begin() + num) << endl;
    return os;
}
```

我们调用 qr.lines 指向的 set 的 size 成员来报告单词出现了多少次。由于 set 是一个 shared_ptr，必须解引用 lines。调用 make_plural（参见 6.3.2 节，第 201 页）来根据大小是否等于 1 打印 time 或 times。

在 for 循环中，我们遍历 lines 所指向的 set。for 循环体打印行号，并按人们习惯的方式调整计数值。set 中的数值就是 vector 中元素的下标，从 0 开始编号。但大多数用户认为第一行的行号应该是 1，因此我们对每个行号都加上 1，转换为人们更习惯的形式。

我们用行号从 file 指向的 vector 中提取一行文本。回忆一下，当给一个迭代器加上一个数时，会得到 vector 中相应偏移之后位置的元素（参见 3.4.2 节，第 99 页）。因此，file->begin()+num 即为 file 指向的 vector 中第 num 个位置的元素。

注意此函数能正确处理未找到单词的情况。在此情况下，set 为空。第一条输出语句会注意到单词出现了 0 次。由于*res.lines 为空，for 循环一次也不会执行。

12.3.2 节练习

练习 12.30：定义你自己版本的 TextQuery 和 QueryResult 类，并执行 12.3.1 节（第 431 页）中的 runQueries 函数。

练习 12.31：如果用 vector 代替 set 保存行号，会有什么差别？哪种方法更好？为什么？

练习 12.32：重写 TextQuery 和 QueryResult 类，用 StrBlob 代替 vector<string> 保存输入文件。

练习 12.33：在第 15 章中我们将扩展查询系统，在 QueryResult 类中将会需要一些额外的成员。添加名为 begin 和 end 的成员，返回一个迭代器，指向一个给定查询返回的行号的 set 中的位置。再添加一个名为 get_file 的成员，返回一个 shared_ptr，指向 QueryResult 对象中的文件。

TextQuery 2 文本查询程序 2

C++ Primer 5th Edition, Chapter 15

634

```
class Bulk_quote : public Quote {
    Bulk_quote* clone() const & {return new Bulk_quote(*this);}
    Bulk_quote* clone() &&
                    {return new Bulk_quote(std::move(*this));}
    // 其他成员与之前的版本一致
};
```

因为我们拥有 add_item 的拷贝和移动版本，所以我们分别定义 clone 的左值和右值版本(参见 13.6.3 节，第 483 页)。每个 clone 函数分配当前类型的一个新对象，其中，const 左值引用成员将它自己拷贝给新分配的对象；右值引用成员则将自己移动到新数据中。

我们可以使用 clone 很容易地写出新版本的 add_item:

```
class Basket {
public:
    void add_item(const Quote& sale)      // 拷贝给定的对象
      { items.insert(std::shared_ptr<Quote>(sale.clone())); }
    void add_item(Quote&& sale)           // 移动给定的对象
      { items.insert(
          std::shared_ptr<Quote>(std::move(sale).clone())); }
    // 其他成员与之前的版本一致
};
```

和 add_item 本身一样，clone 函数也根据作用于左值还是右值而分为不同的重载版本。在此例中，第一个 add_item 函数调用 clone 的 const 左值版本，第二个函数调用 clone 的右值引用版本。在右值版本中，尽管 sale 的类型是右值引用类型，但实际上 sale 本身（和任何其他变量一样）是个左值（参见 13.6.1 节，第 471 页)。因此，我们调用 move 把一个右值引用绑定到 sale 上。

我们的 clone 函数也是一个虚函数。sale 的动态类型(通常)决定了到底运行 Quote 的函数还是 Bulk_quote 的函数。无论我们是拷贝还是移动数据，clone 都返回一个新分配对象的指针，该对象与 clone 所属的类型一致。我们把一个 shared_ptr 绑定到这个对象上，然后调用 insert 将这个新分配的对象添加到 items 中。注意，因为 shared_ptr 支持派生类向基类的类型转换（参见 15.2.2 节，第 530 页)，所以我们能把 shared_ptr<Quote>绑定到 Bulk_quote*上。

15.8.1 节练习

练习 15.30：编写你自己的 Basket 类，用它计算上一个练习中交易记录的总价格。

15.9 文本查询程序再探

635

接下来，我们扩展 12.3 节（第 430 页）的文本查询程序，用它作为说明继承的最后一个例子。在上一版的程序中，我们可以查询在文件中某个指定单词的出现情况。我们将在本节扩展该程序使其支持更多更复杂的查询操作。在后面的例子中，我们将针对下面这个小故事展开查询：

```
Alice Emma has long flowing red hair.
Her Daddy says when the wind blows
through her hair, it looks almost alive,
like a fiery bird in flight.
```

```
A beautiful fiery bird, he tells her,
magical but untamed.
"Daddy, shush, there is no such thing,"
she tells him, at the same time wanting
him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"
```

我们的系统将支持如下查询形式。

- 单词查询，用于得到匹配某个给定 `string` 的所有行：

```
Executing Query for: Daddy
Daddy occurs 3 times
(line 2) Her Daddy says when the wind blows
(line 7) "Daddy, shush, there is no such thing,"
(line 10) Shyly, she asks, "I mean, Daddy, is there?"
```

- 逻辑非查询，使用~运算符得到不匹配查询条件的所有行：

```
Executing Query for: ~(Alice)
~(Alice) occurs 9 times
(line 2) Her Daddy says when the wind blows
(line 3) through her hair, it looks almost alive,
(line 4) like a fiery bird in flight.
...
```

- 逻辑或查询，使用 | 运算符返回匹配两个条件中任意一个的行：

```
Executing Query for: (hair | Alice)
(hair | Alice) occurs 2 times
(line 1) Alice Emma has long flowing red hair.
(line 3) through her hair, it looks almost alive,
```

- 逻辑与查询，使用&运算符返回匹配全部两个条件的行：

```
Executing query for: (hair & Alice)
(hair & Alice) occurs 1 time
(line 1) Alice Emma has long flowing red hair.
```

此外，我们还希望能够混合使用这些运算符，比如：

```
fiery & bird | wind
```

在类似这样的例子中，我们将使用 C++通用的优先级规则（参见 4.1.2 节，第 121 页）对复杂表达式求值。因此，这条查询语句所得行应该是如下二者之一：在该行中或者 `fiery` 和 `bird` 同时出现，或者出现了 `wind`：

```
Executing Query for: ((fiery & bird) | wind)
((fiery & bird) | wind) occurs 3 times
(line 2) Her Daddy says when the wind blows
(line 4) like a fiery bird in flight.
(line 5) A beautiful fiery bird, he tells her,
```

636

在输出内容中首先是那条查询语句，我们使用圆括号来表示查询被解释和执行的次序。与之前实现的版本一样，接下来系统将按照查询结果中行号的升序显示结果并且每一行只显示一次。

15.9.1 面向对象的解决方案

我们可能会认为使用 12.3.2 节（第 432 页）的 `TextQuery` 类来表示单词查询，然后

从该类中派生出其他查询是一种可行的方案。

然而，这样的设计实际上存在缺陷。为了理解其中的原因，我们不妨考虑逻辑非查询。单词查询查找一个指定的单词，为了让逻辑非查询按照单词查询的方式执行，我们将不得不定义逻辑非查询所要查找的单词。但是在一般情况下，我们无法得到这样的单词。相反，一个逻辑非查询中含有一个结果值需要取反的查询语句（单词查询或任何其他查询）；类似的，一个逻辑与查询和一个逻辑或查询各包含两个结果值需要合并的查询语句。

由上述观察结果可知，我们应该将几种不同的查询建模成相互独立的类，这些类共享一个公共基类：

```
WordQuery          // Daddy
NotQuery           // ~Alice
OrQuery            // hair | Alice
AndQuery           // hair & Alice
```

这些类将只包含两个操作：

- `eval`，接受一个 `TextQuery` 对象并返回一个 `QueryResult`，`eval` 函数使用给定的 `TextQuery` 对象查找与之匹配的行。
- `rep`，返回基础查询的 `string` 表示形式，`eval` 函数使用 `rep` 创建一个表示匹配结果的 `QueryResult`，输出运算符使用 `rep` 打印查询表达式。

637

关键概念：继承与组合

继承体系的设计本身是一个非常复杂的问题，已经超出了本书的范围。然而，有一条设计准则非常重要也非常基础，每个程序员都应该熟悉它。

当我们令一个类公有地继承另一个类时，派生类应当反映与基类的“是一种（Is A）”关系。在设计良好的类体系当中，公有派生类的对象应该可以用在任何需要基类对象的地方。

类型之间的另一种常见关系是“有一个（Has A）”关系，具有这种关系的类暗含成员的意思。

在我们的书店示例中，基类表示的是按规定价格销售的书籍的报价。`Bulk_quote` “是一种”报价结果，只不过它使用的价格策略不同。我们的书店类都“有一个”价格成员和 `ISBN` 成员。

抽象基类

如我们所知，在这四种查询之间并不存在彼此的继承关系，从概念上来说它们互为兄弟。因为所有这些类都共享同一个接口，所以我们需要定义一个抽象基类（参见 15.4 节，第 541 页）来表示该接口。我们将所需的抽象基类命名为 `Query_base`，以此来表示它的角色是整个查询继承体系的根节点。

我们的 `Query_base` 类将把 `eval` 和 `rep` 定义成纯虚函数（参见 15.4 节，第 541 页），其他代表某种特定查询类型的类必须覆盖这两个函数。我们将从 `Query_base` 直接派生出 `WordQuery` 和 `NotQuery`。`AndQuery` 和 `OrQuery` 都具有系统中其他类所不具备的一个特殊属性：它们各自包含两个运算对象。为了对这种属性建模，我们定义另外一个名为 `BinaryQuery` 的抽象基类，该抽象基类用于表示含有两个运算对象的查询。`AndQuery` 和 `OrQuery` 继承自 `BinaryQuery`，而 `BinaryQuery` 继承自 `Query_base`。由这些分

析我们将得到如图 15.2 所示的类设计结果：

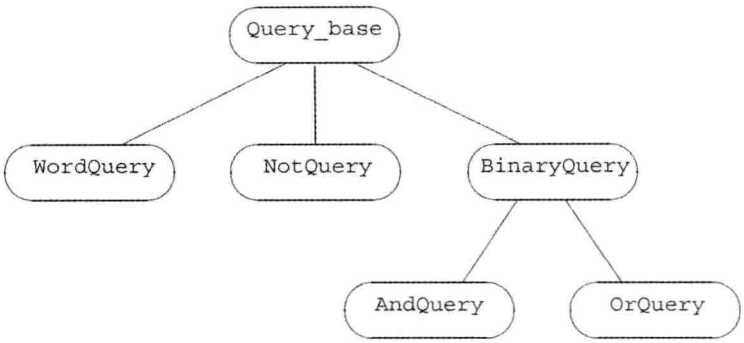

图 15.2：Query_base 继承体系

将层次关系隐藏于接口类中

我们的程序将致力于计算查询结果，而非仅仅构建查询的体系。为了使程序能正常运行，我们必须首先创建查询命令，最简单的办法是编写 C++表达式。例如，可以编写下面的代码来生成之前描述的复合查询：

```
Query q = Query("fiery") & Query("bird") | Query("wind");
```

如上所述，其隐含的意思是用户层代码将不会直接使用这些继承的类；相反，我们将定义一个名为 Query 的接口类，由它负责隐藏整个继承体系。Query 类将保存一个 Query_base 指针，该指针绑定到 Query_base 的派生类对象上。Query 类与 Query_base 类提供的操作是相同的：eval 用于求查询的结果，rep 用于生成查询的 string 版本，同时 Query 也会定义一个重载的输出运算符用于显示查询。 638

用户将通过 Query 对象的操作间接地创建并处理 Query_base 对象。我们定义 Query 对象的三个重载运算符以及一个接受 string 参数的 Query 构造函数，这些函数动态分配一个新的 Query_base 派生类的对象：

- &运算符生成一个绑定到新的 AndQuery 对象上的 Query 对象；
- |运算符生成一个绑定到新的 OrQuery 对象上的 Query 对象；
- ~运算符生成一个绑定到新的 NotQuery 对象上的 Query 对象；
- 接受 string 参数的 Query 构造函数生成一个新的 WordQuery 对象。

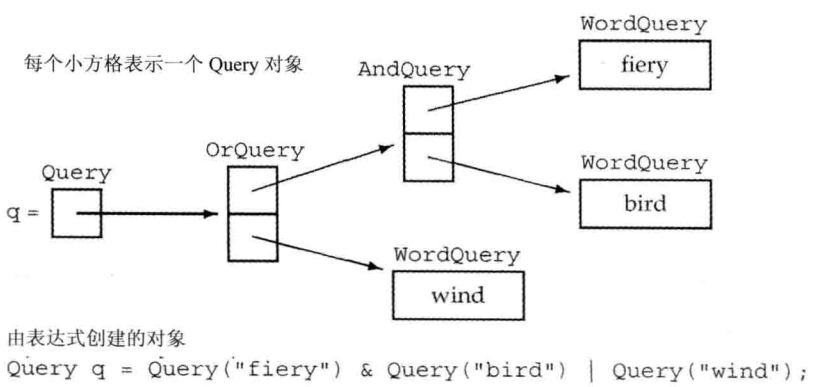

图 15.3：使用 Query 表达式创建的对象

理解这些类的工作机理

在这个应用程序中，很大一部分工作是构建代表用户查询的对象，对于读者来说认识到这一点非常重要。例如，像上面这样的表达式将生成如图 15.3 所示的一系列相关对象的集合。

一旦对象树构建完成后，对某一条查询语句的求值（或生成表示形式的）过程基本上就转换为沿着箭头方向依次对每个对象求值（或显示）的过程（由编译器为我们组织管理）。
639 例如，如果我们对 q（即树的根节点）调用 eval 函数，则该调用语句将令 q 所指的 OrQuery 对象 eval 它自己。对该 OrQuery 求值实际上是对它的两个运算对象执行 eval 操作：一个运算对象是 AndQuery，另一个是查找单词 wind 的 WordQuery。接下来，对 AndQuery 求值转化为对它的两个 WordQuery 求值，分别生成单词 fiery 和 bird 的查询结果。

对于面向对象编程的新手来说，要想理解一个程序，最困难的部分往往是理解程序的设计思路。一旦你掌握了程序的设计思路，接下来的实现也就水到渠成了。为了帮助读者理解程序设计的过程，我们在表 15.1 中整理了之前那个例子用到的类，并对其进行了简要的描述。

640

表 15.1：概述：Query 程序设计

Query 程序接口类和操作	
`TextQuery`	该类读入给定的文件并构建一个查找图。这个类包含一个 query 操作，它接受一个 string 实参，返回一个 QueryResult 对象；该 QueryResult 对象表示 string 出现的行（12.3.2 节，第 432 页）
`QueryResult`	该类保存一个 query 操作的结果（12.3.2 节，第 433 页）
`Query`	是一个接口类，指向 Query_base 派生类的对象
`Query q(s)`	将 Query 对象 q 绑定到一个存放着 string s 的新 WordQuery 对象上
`q1 & q2`	返回一个 Query 对象，该 Query 绑定到一个存放 q1 和 q2 的新 AndQuery 对象上
`q1 | q2`	返回一个 Query 对象，该 Query 绑定到一个存放 q1 和 q2 的新 OrQuery 对象上
`~q`	返回一个 Query 对象，该 Query 绑定到一个存放 q 的新 NotQuery 对象上
Query 程序实现类	
`Query_base`	查询类的抽象基类
`WordQuery`	Query_base 的派生类，用于查找一个给定的单词
`NotQuery`	Query_base 的派生类，查询结果是 Query 运算对象没有出现的行的集合
`BinaryQuery`	Query_base 派生出来的另一个抽象基类，表示有两个运算对象的查询
`OrQuery`	BinaryQuery 的派生类，返回它的两个运算对象分别出现的行的并集
`AndQuery`	BinaryQuery 的派生类，返回它的两个运算对象分别出现的行的交集

15.9.1 节练习

练习 15.31：已知 s1、s2、s3 和 s4 都是 string，判断下面的表达式分别创建了什么样的对象：

```
(a) Query(s1) | Query(s2) & ~ Query(s3);
(b) Query(s1) | (Query(s2) & ~ Query(s3));
(c) (Query(s1) & (Query(s2)) | (Query(s3) & Query(s4)));
```

15.9.2 Query_base 类和 Query 类

下面我们开始程序的实现过程，首先定义 Query_base 类：

```
// 这是一个抽象基类，具体的查询类型从中派生，所有成员都是 private 的
class Query_base {
    friend class Query;
protected:
    using line_no = TextQuery::line_no;   // 用于 eval 函数
    virtual ~Query_base() = default;
private:
    // eval 返回与当前 Query 匹配的 QueryResult
    virtual QueryResult eval(const TextQuery&) const = 0;
    // rep 是表示查询的一个 string
    virtual std::string rep() const = 0;
};
```

eval 和 rep 都是纯虚函数，因此 Query_base 是一个抽象基类（参见 15.4 节，第 541 页）。因为我们不希望用户或者派生类直接使用 Query_base，所以它没有 public 成员。所有对 Query_base 的使用都需要通过 Query 对象，因为 Query 需要调用 Query_base 的虚函数，所以我们将 Query 声明成 Query_base 的友元。

受保护的成员 line_no 将在 eval 函数内部使用。类似的，析构函数也是受保护的，因为它将（隐式地）在派生类析构函数中使用。

Query 类

Query 类对外提供接口，同时隐藏了 Query_base 的继承体系。每个 Query 对象都含有一个指向 Query_base 对象的 shared_ptr。因为 Query 是 Query_base 的唯一接口，所以 Query 必须定义自己的 eval 和 rep 版本。

接受一个 string 参数的 Query 构造函数将创建一个新的 WordQuery 对象，然后将它的 shared_prt 成员绑定到这个新创建的对象上。&、| 和 ~ 运算符分别创建 AndQuery、OrQuery 和 NotQuery 对象，这些运算符将返回一个绑定到新创建的对象上的 Query 对象。为了支持这些运算符，Query 还需要另外一个构造函数，它接受指向 Query_base 的 shared_ptr 并且存储给定的指针。我们将这个构造函数声明为私有的，原因是我们不希望一般的用户代码能随便定义 Query_base 对象。因为这个构造函数是私有的，所以我们需要将三个运算符声明为友元。

在形成了上述设计思路后，Query 类本身就比较简单了：

```
// 这是一个管理 Query_base 继承体系的接口类
class Query {
    // 这些运算符需要访问接受 shared_ptr 的构造函数，而该函数是私有的
    friend Query operator~(const Query &);
    friend Query operator|(const Query&, const Query&);
    friend Query operator&(const Query&, const Query&);
public:                                                    641
    Query(const std::string&);          // 构建一个新的 WordQuery
    // 接口函数：调用对应的 Query_base 操作
    QueryResult eval(const TextQuery &t) const
                            { return q->eval(t); }
    std::string rep() const { return q->rep(); }
```

```
private:
    Query(std::shared_ptr<Query_base> query): q(query) { }
    std::shared_ptr<Query_base> q;
};
```

我们首先将创建 Query 对象的运算符声明为友元，之所以这么做是因为这些运算符需要访问那个私有构造函数。

在 Query 的公有接口部分，我们声明了接受 string 的构造函数，不过没有对其进行定义。因为这个构造函数将要创建一个 WordQuery 对象，所以我们应该首先定义 WordQuery 类，随后才能定义接受 string 的 Query 构造函数。

另外两个公有成员是 Query_base 的接口。其中，Query 操作使用它的 Query_base 指针来调用各自的 Query_base 虚函数。实际调用哪个函数版本将由 q 所指的对象类型决定，并且直到运行时才能最终确定下来。

Query 的输出运算符

输出运算符可以很好地解释我们的整个查询系统是如何工作的：

```
std::ostream &
operator<<(std::ostream &os, const Query &query)
{
    // Query::rep 通过它的 Query_base 指针对 rep()进行了虚调用
    return os << query.rep();
}
```

当我们打印一个 Query 时，输出运算符调用 Query 类的公有 rep 成员。运算符函数通过指针成员虚调用当前 Query 所指对象的 rep 成员。也就是说，当我们编写如下代码时：

```
Query andq = Query(sought1) & Query(sought2);
cout << andq << endl;
```

输出运算符将调用 andq 的 Query::rep，而 Query::rep 通过它的 Query_base 指针虚调用 Query_base 版本的 rep 函数。因为 andq 指向的是一个 AndQuery 对象，所以本次的函数调用将运行 AndQuery::rep。

15.9.2 节练习

练习 15.32：当一个 Query 类型的对象被拷贝、移动、赋值或销毁时，将分别发生什么？

练习 15.33：当一个 Query_base 类型的对象被拷贝、移动、赋值或销毁时，将分别发生什么？

642

15.9.3 派生类

对于 Query_base 的派生类来说，最有趣的部分是这些派生类如何表示一个真实的查询。其中 WordQuery 类最直接，它的任务就是保存要查找的单词。

其他类分别操作一个或两个运算对象。NotQuery 有一个运算对象，AndQuery 和 OrQuery 有两个。在这些类当中，运算对象可以是 Query_base 的任意一个派生类的对象：一个 NotQuery 对象可以被用在 WordQuery、AndQuery、OrQuery 或另一个 NotQuery 中。为了支持这种灵活性，运算对象必须以 Query_base 指针的形式存储，

这样我们就能把该指针绑定到任何我们需要的具体类上。

然而，实际上我们的类并不存储 Query_base 指针，而是直接使用一个 Query 对象。就像用户代码可以通过接口类得到简化一样，我们也可以使用接口类来简化我们自己的类。

至此我们已经清楚了所有类的设计思路，接下来依次实现它们。

WordQuery 类

一个 WordQuery 查找一个给定的 string，它是在给定的 TextQuery 对象上实际执行查询的唯一一个操作：

```
class WordQuery: public Query_base {
    friend class Query;                    // Query 使用 WordQuery 构造函数
    WordQuery(const std::string &s): query_word(s) { }
    // 具体的类：WordQuery 将定义所有继承而来的纯虚函数
    QueryResult eval(const TextQuery &t) const
                    { return t.query(query_word); }
    std::string rep() const { return query_word; }
    std::string query_word;                // 要查找的单词
};
```

和 Query_base 一样，WordQuery 没有公有成员。同时，Query 必须作为 WordQuery 的友元，这样 Query 才能访问 WordQuery 的构造函数。

每个表示具体查询的类都必须定义继承而来的纯虚函数 eval 和 rep。我们在 WordQuery 类的内部定义这两个操作：eval 调用其 TextQuery 参数的 query 成员，由 query 成员在文件中实际进行查找；rep 返回这个 WordQuery 表示的 string（即 query_word）。

定义了 WordQuery 类之后，我们就能定义接受 string 的 Query 构造函数了：

```
inline
Query::Query(const std::string &s): q(new WordQuery(s)) { }
```

这个构造函数分配一个 WordQuery，然后令其指针成员指向新分配的对象。

NotQuery 类及~运算符

~运算符生成一个 NotQuery，其中保存着一个需要对其取反的 Query：

```
class NotQuery: public Query_base {
    friend Query operator~(const Query &);
    NotQuery(const Query &q): query(q) { }
    // 具体的类：NotQuery 将定义所有继承而来的纯虚函数
    std::string rep() const {return "~(" + query.rep() + ")";}
    QueryResult eval(const TextQuery&) const;
    Query query;
};
inline Query operator~(const Query &operand)
{
    return std::shared_ptr<Query_base>(new NotQuery(operand));
}
```

643

因为 NotQuery 的所有成员都是私有的，所以我们一开始就要把~运算符设定为友元。为

了 rep 一个 NotQuery，我们需要将~符号与基础的 Query 连接在一起。我们在输出的结果中加上适当的括号，这样读者就可以清楚地知道查询的优先级了。

值得注意的是，在 NotQuery 自己的 rep 成员中对 rep 的调用最终执行的是一个虚调用：query.rep()是对 Query 类 rep 成员的非虚调用，接着 Query::rep 将调用 q->rep()，这是一个通过 Query_base 指针进行的虚调用。

~运算符动态分配一个新的 NotQuery 对象，其 return 语句隐式地使用接受一个 shared_ptr<Query_base>的 Query 构造函数。也就是说，return 语句等价于：

```
// 分配一个新的 NotQuery 对象
// 将所得的 NotQuery 指针绑定到一个 shared_ptr<Query_base>
shared_ptr<Query_base> tmp(new NotQuery(expr));
return Query(tmp);          // 使用接受一个 shared_ptr 的 Query 构造函数
```

eval 成员比较复杂，因此我们将在类的外部实现它，15.9.4 节（第 573 页）将专门介绍如何定义 eval 函数。

BinaryQuery 类

BinaryQuery 类也是一个抽象基类，它保存操作两个运算对象的查询类型所需的数据：

```
class BinaryQuery: public Query_base {
protected:
    BinaryQuery(const Query &l, const Query &r, std::string s):
          lhs(l), rhs(r), opSym(s) { }
    // 抽象类：BinaryQuery 不定义 eval
    std::string rep() const { return "(" + lhs.rep() + " "
                                         + opSym + " "
                                         + rhs.rep() + ")"; }
    Query lhs, rhs;           // 左侧和右侧运算对象
    std::string opSym;        // 运算符的名字
};
```

644 BinaryQuery 中的数据是两个运算对象及相应的运算符符号，构造函数负责接受两个运算对象和一个运算符符号，然后将它们存储在对应的数据成员中。

要想 rep 一个 BinaryQuery，我们需要生成一个带括号的表达式。表达式的内容依次包括左侧运算对象、运算符以及右侧运算对象。就像我们显示 NotQuery 的方法一样，对 rep 的调用最终是对 lhs 和 rhs 所指 Query_base 对象的 rep 函数进行虚调用。

BinaryQuery 不定义 eval，而是继承了该纯虚函数。因此，BinaryQuery 也是一个抽象基类，我们不能创建 BinaryQuery 类型的对象。

AndQuery 类、OrQuery 类及相应的运算符

AndQuery 类和 OrQuery 类以及它们的运算符都非常相似：

```
class AndQuery: public BinaryQuery {
    friend Query operator&(const Query&, const Query&);
    AndQuery(const Query &left, const Query &right):
                        BinaryQuery(left, right, "&") { }
    // 具体的类：AndQuery 继承了 rep 并且定义了其他纯虚函数
    QueryResult eval(const TextQuery&) const;
```

```
};
inline Query operator&(const Query &lhs, const Query &rhs)
{
    return std::shared_ptr<Query_base>(new AndQuery(lhs, rhs));
}

class OrQuery: public BinaryQuery {
    friend Query operator|(const Query&, const Query&);
    OrQuery(const Query &left, const Query &right):
                BinaryQuery(left, right, "|") { }
    QueryResult eval(const TextQuery&) const;
};
inline Query operator|(const Query &lhs, const Query &rhs)
{
    return std::shared_ptr<Query_base>(new OrQuery(lhs, rhs));
}
```

这两个类将各自的运算符定义成友元，并且各自定义了一个构造函数通过运算符创建 BinaryQuery 基类部分。它们继承 BinaryQuery 的 rep 函数，但是覆盖了 eval 函数。

和~运算符一样，&和|运算符也返回一个绑定到新分配对象上的 shared_ptr。在这些运算符中，return 语句负责将 shared_ptr 转换成 Query。

15.9.3 节练习

645

练习 15.34：针对图 15.3（第 565 页）构建的表达式：

(a) 列举出在处理表达式的过程中执行的所有构造函数。
(b) 列举出 cout<<q 所调用的 rep。
(c) 列举出 q.eval()所调用的 eval。

练习 15.35：实现 Query 类和 Query_base 类，其中需要定义 rep 而无须定义 eval。

练习 15.36：在构造函数和 rep 成员中添加打印语句，运行你的代码以检验你对本节第一个练习中（a）、（b）两小题的回答是否正确。

练习 15.37：如果在派生类中含有 shared_ptr<Query_base>类型的成员而非 Query 类型的成员，则你的类需要做出怎样的改变？

练习 15.38：下面的声明合法吗？如果不合法，请解释原因；如果合法，请指出该声明的含义。

```
BinaryQuery a = Query("fiery") & Query("bird");
AndQuery b = Query("fiery") & Query("bird");
OrQuery c = Query("fiery") & Query("bird");
```

15.9.4 eval 函数

eval 函数是我们这个查询系统的核心。每个 eval 函数作用于各自的运算对象，同时遵循的内在逻辑也有所区别：OrQuery 的 eval 操作返回两个运算对象查询结果的并集，而 AndQuery 返回交集。与它们相比，NotQuery 的 eval 函数更加复杂一些：它需要返回运算对象没有出现的文本行。

为了支持上述 eval 函数的处理，我们需要使用 QueryResult，在它当中定义了 12.3.2 节练习（第 435 页）添加的成员。假设 QueryResult 包含 begin 和 end 成员，它们允许我们在 QueryResult 保存的行号 set 中进行迭代；另外假设 QueryResult 还包含一个名为 get_file 的成员，它返回一个指向待查询文件的 shared_ptr。

我们的 Query 类使用了 12.3.2 节练习（第 435 页）为 QueryResult 定义的成员。

OrQuery::eval

一个 OrQuery 表示的是它的两个运算对象结果的并集，对于每个运算对象来说，我们通过调用 eval 得到它的查询结果。因为这些运算对象的类型是 Query，所以调用 eval 也就是调用 Query::eval，而后者实际上是对潜在的 Query_base 对象的 eval 进行虚调用。每次调用完成后，得到的结果是一个 QueryResult，它表示运算对象出现的行号。我们把这些行号组织在一个新 set 中：

646

```
// 返回运算对象查询结果 set 的并集
QueryResult
OrQuery::eval(const TextQuery& text) const
{
    // 通过 Query 成员 lhs 和 rhs 进行的虚调用
    // 调用 eval 返回每个运算对象的 QueryResult
    auto right = rhs.eval(text), left = lhs.eval(text);
    // 将左侧运算对象的行号拷贝到结果 set 中
    auto ret_lines =
        make_shared<set<line_no>>(left.begin(), left.end());
    // 插入右侧运算对象所得的行号
    ret_lines->insert(right.begin(), right.end());
    // 返回一个新的 QueryResult，它表示 lhs 和 rhs 的并集
    return QueryResult(rep(), ret_lines, left.get_file());
}
```

我们使用接受一对迭代器的 set 构造函数初始化 ret_lines。一个 QueryResult 的 begin 和 end 成员返回行号 set 的迭代器，因此，创建 ret_lines 的过程实际上是拷贝了 left 集合的元素。接下来对 ret_lines 调用 insert，并将 right 的元素插入进来。调用结束后，ret_lines 将包含在 left 或 right 中出现过的所有行号。

eval 函数在最后构建并返回一个表示混合查询匹配的 QueryResult。QueryResult 的构造函数（参见 12.3.2 节，第 434 页）接受三个实参：一个表示查询的 string、一个指向匹配行号 set 的 shared_ptr 和一个指向输入文件 vector 的 shared_ptr。我们调用 rep 生成所需的 string，调用 get_file 获取指向文件的 shared_ptr。因为 left 和 right 指向的是同一个文件，所以使用哪个执行 get_file 函数并不重要。

AndQuery::eval

AndQuery 的 eval 和 OrQuery 很类似，唯一的区别是它调用了一个标准库算法来求得两个查询结果中共有的行：

```
// 返回运算对象查询结果 set 的交集
QueryResult
AndQuery::eval(const TextQuery& text) const
{
```

```
    // 通过 Query 运算对象进行的虚调用，以获得运算对象的查询结果 set
    auto left = lhs.eval(text), right = rhs.eval(text);
    // 保存 left 和 right 交集的 set
    auto ret_lines = make_shared<set<line_no>>();
    // 将两个范围的交集写入一个目的迭代器中
    // 本次调用的目的迭代器向 ret 添加元素
    set_intersection(left.begin(), left.end(),
                     right.begin(), right.end(),
                     inserter(*ret_lines, ret_lines->begin()));
    return QueryResult(rep(), ret_lines, left.get_file());
}
```

其中我们使用标准库算法 set_intersection 来合并两个 set，关于 set_intersection 在附录 A.2.8（第 779 页）中有详细的描述。 647

set_intersection 算法接受五个迭代器。它使用前四个迭代器表示两个输入序列（参见 10.5.2 节，第 368 页），最后一个实参表示目的位置。该算法将两个输入序列中共同出现的元素写入到目的位置中。

在上述调用中我们传入一个插入迭代器（参见 10.4.1 节，第 357 页）作为目的位置。当 set_intersection 向这个迭代器写入内容时，实际上是向 ret_lines 插入一个新元素。

和 OrQuery 的 eval 函数一样，AndQuery 的 eval 函数也在最后构建并返回一个表示混合查询匹配的 QueryResult。

NotQuery::eval

NotQuery 查找运算对象没有出现的文本行：

```
// 返回运算对象的结果 set 中不存在的行
QueryResult
NotQuery::eval(const TextQuery& text) const
{
    // 通过 Query 运算对象对 eval 进行虚调用
    auto result = query.eval(text);
    // 开始时结果 set 为空
    auto ret_lines = make_shared<set<line_no>>();
    // 我们必须在运算对象出现的所有行中进行迭代
    auto beg = result.begin(), end = result.end();
    // 对于输入文件的每一行，如果该行不在 result 当中，则将其添加到 ret_lines
    auto sz = result.get_file()->size();
    for (size_t n = 0; n != sz; ++n) {
        // 如果我们还没有处理完 result 的所有行
        // 检查当前行是否存在
        if (beg == end || *beg != n)
            ret_lines->insert(n);     // 如果不在 result 当中，添加这一行
        else if (beg != end)
            ++beg;          // 否则继续获取 result 的下一行（如果有的话）
    }
    return QueryResult(rep(), ret_lines, result.get_file());
}
```

和其他 eval 函数一样，我们首先对当前的运算对象调用 eval，所得的结果

QueryResult 中包含的是运算对象出现的行号，但我们想要的是运算对象未出现的行号。也就是说，我们需要的是存在于文件中，但是不在 result 中的行。

要想得到最终的结果，我们需要遍历不超过输出文件大小的所有整数，并将所有不在 result 中的行号放入到 ret_lines 中。我们使用 beg 和 end 分别表示 result 的第一个元素和最后一个元素的下一位置。因为遍历的对象是一个 set，所以当遍历结束后获得的行号将按照升序排列。

648

循环体负责检查当前的编号是否在 result 当中。如果不在，将这个数字添加到 ret_lines 中；如果该数字属于 result，则我们递增 result 的迭代器 beg。

一旦处理完所有行号，就返回包含 ret_lines 的一个 QueryResult 对象；和之前版本的 eval 类似，该 QueryResult 对象还包含 rep 和 get_file 的运行结果。

15.9.4 节练习

练习 15.39：实现 Query 类和 Query_base 类，求图 15.3（第 565 页）中表达式的值并打印相关信息，验证你的程序是否正确。

练习 15.40：在 OrQuery 的 eval 函数中，如果 rhs 成员返回的是空集将发生什么？如果 lhs 是空集呢？如果 lhs 和 rhs 都是空集又将发生什么？

练习 15.41：重新实现你的类，这次使用指向 Query_base 的内置指针而非 shared_ptr。请注意，做出上述改动后你的类将不能再使用合成的拷贝控制成员。

练习 15.42：从下面的几种改进中选择一种，设计并实现它：

(a) 按句子查询并打印单词，而不再是按行打印。

(b) 引入一个历史系统，用户可以按编号查阅之前的某个查询，并可以在其中增加内容或者将其与其他查询组合。

(c) 允许用户对结果做出限制，比如从给定范围的行中挑出匹配的进行显示。