第 4 章　C++的扩展

对于典型的C++程序员，程序设计不只是工作，还是一种生活方式。程序员不只是需要语言的原始功能，还需要它的细微与精妙。就像一个品酒师品味葡萄酒一样，我们也享受语言出色的特征。当然，我们对C++的兴趣不会只局限于用它来编写程序。相反，通常我们发现自己被语言本身以及围绕语言的设计和开发的问题所吸引。

大多数C++程序员都对计算机语言有兴趣，很少有C++程序员没有梦想过对语言加入新的功能。(您想过多少次“如果 C++能够……该多好啊”？)问题是大多数程序员都不能够访问完善的C++编译器来加入他们的实验性结构。幸运的是，在此有一个简单的方法来试验您自己对C++的扩展：通过创建一个译码器(translator)，将您的想法转换为实际的C++代码。这种译码器是本章的主题。

4.1　为什么使用译码器

由于在此有一个明显的选择：选择标准C++预处理器，您可能会问，为什么需要使用译码器来体验新的语言特性。大家都知道，预处理器支持宏替换，其中可以用一个文本序列来替换另一个文本序列。许多年来，预处理器宏一直被用来向C++中加入新的特性。例如，宏经常实现的一个结构是repeat/until循环。repeat/until循环类似于C++的do/while循环，只是repeat/until循环只有在条件变为true时才会终止。(也就是说，repeat/until在条件为false时运行)。这不同于do/while循环，它在条件变为false时终止。由于repeat/until循环和do/while循环之间的相似性，可以用宏来实现repeat/until循环，如下所示：

```
#define repeat do
#define until(exp) while(!(exp))
// ...
int i=0;
repeat {
  cout << "i: " << i << endl;
  i++;
} until(i==10);
```

这个宏使得repeat取代do，　until取代while。另外，在while中的条件表达式被反转。因此，预处理器会将前面代码段中的repeat/until循环转换为这个do/while循环：

```
do {
  cout << "i: " << i << endl;
  i++;
} while(!(i==10));
```

正如这个示例显示的，使用宏可以轻松合理地实现repeat/until结构。

既然使用宏实现新的特性已经有好多年的历史，那为什么要创建译码器呢？也就是说，为什么不像刚才对 repeat/until 那样使用宏呢？答案是，并不能使用宏替换来加入所有类型的新特性。另外，即使可以使用宏替换，也并不是所有的替换都像 repeat/until 示例那样优雅并且在概念上很完美。尽管程序员已经通过宏实现了很多令人瞩目的特性，但是这些任务在很多情况下是通过使用复杂的、难以理解的#define 指示来完成的。这种“宏陷阱”经常会导致结构不好的代码，很难验证其正确性，并且缺乏弹性。坦白地说，由于上面的原因，许多 C++程序员(包括作者)避免使用复杂的宏。

幸运的是，在此有一个对复杂的宏的替代方案，可以用它试验多种新的 C++特性：可以创建一个译码器，在程序的控制之下将试验结构转换为等价的 C++代码。使用这种“提前预处理器”可以向 C++加入使用预处理器宏很难实现甚至不可能实现的特性。这个译码器本身就是一个有趣的项目，并且可以很容易地修改这个框架来适应其他需要。

尽管本章开发的译码器能够执行复杂的文本替换，但它仍然有其局限性，理解这一点很重要。由于这个译码器实现了单向算法，只读取一次源文件，因此它只能用于单向替换就可以处理的结构。(如果您愿意的话，可以修改这个特性)。除了执行复杂的文本替换之外，译码器完全忽略程序的内容。也就是说，它对变量的类型、运算符的意义，甚至先前读取的内容一无所知。因此，这个译码器并不是 C++语言的分析器。它只是对指定文本替换的一个引擎；在本质上，它是一个特殊的预处理器。尽管有这些限制，仍然可以使用它试验各种各样的想法。

提示：

如果您对 C++的分析器感兴趣，请参考第 9 章。

4.2 实验性的关键字

在开发译码器之前，有必要定义它所执行的转换。本章开发的译码器处理如下的对 C++实验性的扩展：

- 一个 foreach 循环
- 一条 cases 语句
- 一个 typeof 运算符
- 一个 repeat/until 循环

除了 repeat/until 循环之外(包含它只是为了演示)，其他的关键字都使用了宏替换很难(甚至不可能)转换的语法。对每个实验性关键字的描述如下。

4.2.1 foreach 循环

当前开发的语言已经包含了 foreach 循环。例如，foreach 是 C#的一部分，Java 中也加入了“for each”风格的循环。另外，“for each”的思想已经被整合为最新 C++的一部分，因为 STL(标准模板库)定义了 for_each()算法，对容器中的每个元素都应用了一个函数。然而，C++目前还没有定义多用途的 foreach 循环。

foreach 用来处理程序设计中经常遇到的情况：需要按照严格的顺序从头到尾遍历数组的元素(或者其他类型的对象集合)。例如，考虑这个计算数组元素平均值的代码段：

```
int n[] = { 1, 7, 3, 11, 5 };
double avg=0.0;

for(int i=0; i < 5; i++)
  avg += n[i];

cout << "Average: " << avg/5 << endl;
```

为了计算 n 的平均值，按顺序从头到尾读取了数组的每个元素。当然，这只是普遍概念的一个实例。可以使用相似类型的循环来查找数组的最大值或者最小值、对数组的值求和、计算最小公约数，可能还有上百个其他的用途。另外，当需要访问数组的内容时，都可以使用相同类型的循环结构。foreach 循环就是为了简化并改进这类循环而创建的。

foreach 的价值在于它取消了手工索引数组的需要。相反，foreach 自动遍历整个数组，按顺序在一个时刻获取一个元素。例如，下面是用 foreach 修改过的前面的代码段：

```
int n[] = { 1, 7, 3, 11, 5 };
double avg=0.0;

foreach(int x in n)
  avg += x;

cout << "Average: " << avg/5 << endl;
```

每循环一次，自动赋予 x 的值等于 n 中下一个元素的值。因此，第一次迭代 x 的值为 1，第二次迭代 x 的值为 7，以此类推。从而不仅对语法进行了改进，同时也防止了计数错误。

foreach 循环的语法如下：

```
foreach(type varname in arrayname)
```

在此，type 为循环变量的类型，其名称由 varname 指定，被访问的数组由 arrayname 指定。另外，varname 局限于循环之内，在循环外不会被知道(这个语法由 C#而来)。记住，循环的每次迭代过程中，varname 都会包含(按顺序)指定数组的下一个元素的值。

译码器将 foreach 循环转换为其等价的 C++ for 循环。

4.2.2 cases 语句

cases 语句允许您指定一个值的范围来与 switch 表达式匹配。通常，当您想让两个或者多个 case 语句使用相同的代码序列时，必须使用“堆砌的”case 语句，下面的示例显示了这一点：

```
switch(i) {
  case 1:
  case 2:
  case 3:
  case 4:
    // do something for cases 1 to 4
    break;
  case 5:
    // do something else
    break;
  // ...
}
```

堆砌的 case 语句(如这个示例中的 1 到 4 条)在程序设计中很常见。这是乏味的代码。为什么不在一个 case 内指定一个值的范围来简化呢？译码器正好可以让您做到这一点！

译码器实现了一个 cases 语句，使您可以定义一个值的范围，在一个代码序列中处理范围内的所有值。例如，先前的代码可以使用 cases 语句来修改：

```
switch(i) {
  cases 1 to 4:
    // do something for cases 1 to 4
    break;
  case 5:
    // do something else
    break;
  // ...
}
```

这样，当 i 包括在 1 到 4 之间时，这个值将由 cases 语句来匹配。

cases 语句的语法如下所示：

```
cases start to end:
```

在此，start 是用来匹配的开始值，end 是用来匹配的结束值。

译码器将 cases 语句转换为一系列堆砌的 case 语句。

4.2.3 typeof 运算符

运行时类型 ID 已经成为大多数现代程序设计重要的一部分。尽管 C++内建的对它的支持很不错，但是程序员仍然不停地试图寻求更好的性能。typeof 实验性运算符就是一个例子。它只是提供了 C++已经支持的操作的另一个可选择的语法：两个类型的比较。因此，typeof 没有增加新的功能，但是它提供了这个过程的不同观点。

通常，当需要类型比较时，会使用 typeid 运算符。例如，下面的语句用来判断 ptr1 所指的对象是否与 ptr2 指的对象具有相同的类型：

```
if(typeid(*ptr1) == typeid(*ptr2))
  cout << "ptr1 points to same type as ptr2\n";
```

在这条语句中，typeid 运算符获取了 ptr1 和 ptr2 所指对象的类型。如果这两个类型相同，则 if 语句成功。当使用多态性类时，可能会用到这样的语句。在此情况下，基类指针所指对象的类型不能在编译时得知，从而需要运行时检查。

尽管 typeid 或前面的语句没有错误，但是下面的方法提供了一种更为有趣的可选语法：

```
if(typeof *ptr1 same as *ptr2)
  cout << "ptr1 points to same type as ptr2\n";
```

在这条语句中，typeof 运算符用来比较两种类型。如果这两种类型相同，则返回 true；否则返回 false。尽管它执行的操作与第一个版本相同，但它改变了操作的表达方式，从而使得您以不同的观点来看待它。它还说明了译码器让您可以体验的想法的范围。

typeof 的语法如下：

```
typeof op1 same as op2
```

在此，*op1* 和 *op2* 指定了类型标识符(如 int 或者 MyClass)或者对象。因此，typeof 可以用来比较两个对象的类型，一个对象的类型和一个已知类型，或者两个类型。

译码器将 typeof 转换为其对应的 typeid 表达式。

4.2.4 repeat/until 循环

如前所述，很容易使用预处理器宏来实现 repeat/until 循环。用译码器来实现 repeat/until 只是为了说明问题，并且由于它可以作为其他类型循环的模型，您或许想要试验这些新的循环。

4.3 试验 C++新特性的译码器

译码器的全部代码如下所示。为了前后一致，将这个文件命名为 trans.cpp。

```
// A translator for experimental C++ extensions.
#include <iostream>
#include <fstream>
#include <cctype>
#include <cstring>
#include <string>

using namespace std;

// Prototypes for the functions that handle
// the extended keywords.
void foreach();
void cases();
void repeat();
void until();
void typeof();

// Prototypes for tokenizing the input file.
bool gettoken(string &tok);
void skipspaces();

// Indentation padding string.
string indent = "";

// The input and output file streams.
ifstream fin;
ofstream fout;

// Exception class for syntax errors.
class SyntaxExc {
  string what;
public:
  SyntaxExc(char *e) { what = string(e); }
  string geterror() { return what; }
};
```

```
int main(int argc, char *argv[]) {
  string token;

  if(argc != 3) {
    cout << "Usage: ep <input file> <output file>\n";
    return 1;
  }

  fin.open(argv[1]);

  if(!fin) {
    cout << "Cannot open " << argv[1] << endl;
    return 1;
  }

  fout.open(argv[2]);

  if(!fout) {
    cout << "Cannot open " << argv[2] << endl;
    return 1;
  }

  // Write header.
  fout << "// Translated from an .exp source file.\n";

  try {
    // Main translation loop.
    while(gettoken(token)) {

      // Skip over // comments.
      if(token == "//") {
        do {
          fout << token;
          gettoken(token);
        } while(token.find('\n') == string::npos);
        fout << token;
      }

      // Skip over /* comments.
      else if(token == "/*") {
        do {
          fout << token;
          gettoken(token);
        } while(token != "*/");
        fout << token;
      }

      // Skip over quoted string.
      else if(token == "\"") {
        do {
          fout << token;
```

```
        gettoken(token);
      } while(token != "\"");
      fout << token;
    }

    else if(token == "foreach") foreach();

    else if(token == "cases") cases();

    else if(token == "repeat") repeat();

    else if(token == "until") until();

    else if(token == "typeof") typeof();

    else fout << token;
  }
 } catch(SyntaxExc exc) {
  cout << exc.geterror() << endl;
  return 1;
 }

 return 0;
}

// Get the next token from the input stream.
bool gettoken(string &tok) {
 char ch;
 char ch2;
 static bool trackIndent = true;

 tok = "";

 ch = fin.get();

 // Check for EOF and return false if EOF
 // is found.
 if(!fin) return false;

 // Read whitespace.
 if(isspace(ch)) {
  while(isspace(ch)) {
    tok += ch;

    // Reset indent counter with each new line.
    if(ch == '\n') {
      indent = "";
      trackIndent = true;
    }
    else if(trackIndent) indent += ch;

    ch = fin.get();
```

```
  }
  fin.putback(ch);
  return true;
}

// Stop tracking indentation after encountering
// first non-whitespace character on a line.
trackIndent = false;

// Read an identifier or keyword.
if(isalpha(ch) || ch=='_') {
  while(isalpha(ch) || isdigit(ch) || ch=='_') {
    tok += ch;
    ch = fin.get();
  }
  fin.putback(ch);
  return true;
}

// Read a number.
if(isdigit(ch)) {
  while(isdigit(ch) || ch=='.' ||
        tolower(ch) == 'e' ||
        ch == '-' || ch =='+') {
    tok += ch;
    ch = fin.get();
  }
  fin.putback(ch);
  return true;
}

// Check for \"
if(ch == '\\') {
  ch2 = fin.get();
  if(ch2 == '"') {
    tok += ch;
    tok += ch2;
    ch = fin.get();
  } else
    fin.putback(ch2);
}

// Check for '"'
if(ch == '\'') {
  ch2 = fin.get();
  if(ch2 == '"') {
    tok += ch;
    tok += ch2;
    return true;
  } else
    fin.putback(ch2);
}
```

```
// Check for begin comment symbols.
if(ch == '/') {
  tok += ch;
  ch = fin.get();
  if(ch == '/' || ch == '*') {
    tok += ch;
  }
  else fin.putback(ch);
  return true;
}

// Check for end comment symbols.
if(ch == '*') {
  tok += ch;
  ch = fin.get();
  if(ch == '/') {
    tok += ch;
  }
  else fin.putback(ch);
  return true;
}

tok += ch;
  return true;
}

// Translate a foreach loop.
void foreach() {
string token;
string varname;
string arrayname;

char forvarname[5] = "_i";
static char counter[2] = "a";

// Create loop control variable for generated
// for loop.
strcat(forvarname, counter);
counter[0]++;

// Only 26 foreach loops in a file because
// generated loop control variables limited to
// _ia to _iz. This can be changed if desired.
if(counter[0] > 'z')
  throw SyntaxExc("Too many foreach loops.");

fout << "int " << forvarname
     << " = 0;\n";

// Write beginning of generated for loop.
fout << indent << "for(";
```

```
  skipspaces();

  // Read the (
  gettoken(token);
  if(token[0] != '(')
    throw SyntaxExc("( expected in foreach.");

  skipspaces();

  // Get the type of the foreach variable.
  gettoken(token);
  fout << token << " ";

  skipspaces();

  // Read and save the foreach variable's name.
  gettoken(token);
  varname = token;

  skipspaces();

  // Read the "in"
  gettoken(token);
  if(token != "in")
    throw SyntaxExc("in expected in foreach.");

  skipspaces();

  // Read the array name.
  gettoken(token);
  arrayname = token;

  fout << varname << " = " << arrayname << "[0];\n";

  // Construct target value.
  fout << indent + " " << forvarname << " < "
       << "((sizeof " << token << ")/"
       << "(sizeof " << token << "[0]));\n";

  fout << indent + " " << forvarname << "++, "
       << varname << " = " << arrayname << "["
       << forvarname << "])";

  skipspaces();

  // Read the )
  gettoken(token);
  if(token[0] != ')')
    throw SyntaxExc(") expected in foreach.");
}
```

```
// Translate a cases statement.
void cases() {
  string token;
  int start, end;

  skipspaces();

  // Get starting value.
  gettoken(token);
  if(isdigit(token[0])) {
    // is an int constant
    start = atoi(token.c_str());
  }
  else if(token[0] == '\'') {
    // is char constant
    gettoken(token);

    start = (int) token[0];

    // discard closing '
    gettoken(token);
    if(token[0] != '\'')
      throw SyntaxExc("' expected in cases.");
  }
  else
    throw SyntaxExc("Constant expected in cases.");

  skipspaces();

  // Read and discard the "to".
  gettoken(token);
  if(token != "to")
    throw SyntaxExc("to expected in cases.");

  skipspaces();

  // Get ending value.
  gettoken(token);

  if(isdigit(token[0])) {
    // is an int constant
    end = atoi(token.c_str());
  }
  else if(token[0] == '\'') {
    // is char constant
    gettoken(token);

    end = (int) token[0];

    // discard closing '
    gettoken(token);
    if(token[0] != '\'')
```

```
      throw SyntaxExc("' expected in cases.");
    }
    else
      throw SyntaxExc("Constant expected in cases.");

    skipspaces();

    // Read and discard the :
    gettoken(token);

    if(token != ":")
      throw SyntaxExc(": expected in cases.");

    // Generate stacked case statments.
    fout << "case " << start << ":\n";
    for(int i = start+1 ; i <= end; i++) {
      fout << indent << "case " << i << ":";
      if(i != end) fout << endl;
    }
}

// Translate a repeat loop.
void repeat() {
  fout << "do";
}

// Translate an until.
void until() {
  string token;
  int parencount = 1;

  fout << "while";

  skipspaces();

  // Read and store the (
  gettoken(token);
  if(token != "(")
    throw SyntaxExc("( expected in typeof.");
  fout << "(";

  // Begin while by reversing and
  // parenthesizing the condition.
  fout << "!(";

  // Now, read the expression.
  do {
    if(!gettoken(token))
      throw SyntaxExc("Unexpected EOF encountered.");

    if(token == "(") parencount++;
    if(token == ")") parencount--;
```

```
    fout << token;
  } while(parencount > 0);
  fout << ")";
}

// Translate a typeof expression.
void typeof() {
  string token;
  string temp;

  fout << "typeid(";

  skipspaces();

  gettoken(token);

  do {
    temp = token;

    if(!gettoken(token))
      throw SyntaxExc("Unexpected EOF encountered.");

    if(token != "same") fout << temp;
  } while(token != "same");

  skipspaces();

  gettoken(token);

  if(token != "as") throw SyntaxExc("as expected.");

  fout << ") == typeid(";

  skipspaces();

  do {
    if(!gettoken(token))
      throw SyntaxExc("Unexpected EOF encountered.");

    fout << token;
  } while(token != ")");
  fout << ")";
}

void skipspaces() {
  char ch;

  do {
    ch = fin.get();
  } while(isspace(ch));
  fin.putback(ch);
}
```

4.4 使用译码器

为了使用译码器，首先要创建一个文件，其中包含使用了实验性关键字的程序。为了直观起见，将使用了实验性结构的文件的扩展名定为.exp。需要理解的是，exp 文件主要包含标准C++代码。它只是还包含了一个或者多个实验性的特征，这些特征会被译码器转换为 C++代码。例如，下面是用来说明 foreach 循环的 exp 文件。注意大部分程序还是由普通的 C++代码组成。

```
// Demonstrate the foreach loop.
#include <iostream>
using namespace std;

int main() {
  int nums[] = { 1, 6, 19, 4, -10, 88 };
  int min;

  // Find the minimum value.
  min = nums[0];
  foreach(int x in nums)
    if(min > x) min = x;

  cout << "Minimum is " << min << endl;

  return 0;
}
```

为了将 exp 文件转换为标准的 cpp 文件，要通过译码器运行它。例如，假定这个文件的名称为 foreach.exp，下面的命令行将把它转化为可以被 C++编译器编译的纯粹的 C++代码：

```
trans foreach.exp foreach.cpp
```

在 trans 运行之后，foreach.cpp 将包含如下的 C++程序：

```
// Translated from an .exp source file.
// Demonstrate the foreach loop.
#include <iostream>
using namespace std;

int main() {
  int nums[] = { 1, 6, 19, 4, -10, 88 };
  int min;

  // Find the minimum value.
  min = nums[0];
  int _ia = 0;
  for(int x = nums[0];
      _ia < ((sizeof nums)/(sizeof nums[0]));
      _ia++, x = nums[_ia])
    if(min > x) min = x;

  cout << "Minimum is " << min << endl;
```

```
  return 0;
}
```

本章的剩余部分将讨论译码器的运行方式。

4.5 译码器的运行方式

译码器以直接的方式操作。它只是读取输入文件并将之编写为输出文件。在此过程中，当发现实验性关键字时，就会将其转换为等价的C++代码。这种内在的简单性就是译码器适合于试验的原因。不需要花很大的力气来实现一个扩展，然后再测试它。下面的几节将详细讨论译码器的各个部分。

4.5.1 全局声明

译码器在开始定义了如下的全局变量和类：

```
// Indentation padding string.
string indent = "";
// The input and output file streams.
ifstream fin;
ofstream fout;

// Exception class for syntax errors.
class SyntaxExc {
  string what;
public:
  SyntaxExc(char *e) { what = string(e); }
  string geterror() { return what; }
};
```

当前用来缩进的空白序列存储在 indent 中。当为试验结构替换多行代码时，这个字符串用来加入数量合适的缩进。

输入文件流存储在 fin 中；输出文件流保存在 fout 中。当程序开始时，命令行指定的文件名与 fin 和 fout 链接。

如果在试验结构的转换中出现了语法错误，就会抛出 SyntaxExc 对象来报告。如同它的字面意思那样，SyntaxExc 值包含了描述错误的字符串，但如果需要的话，可以加入新的功能。

4.5.2 main()函数

main()函数执行两个任务。首先，它打开命令行指定的输入和输出文件。C++程序员应该很熟悉这些代码。其次，它运行译码器的主循环，这个循环处理试验关键字的转换。主循环如下所示：

```
try {
  // Main translation loop.
  while(gettoken(token)) {
```

```
    // Skip over // comments.
    if(token == "//") {
      do {
        fout << token;
        gettoken(token);
      } while(token.find('\n') == string::npos);
      fout << token;
    }

    // Skip over /* comments.
    else if(token == "/*") {
      do {
        fout << token;
        gettoken(token);
      } while(token != "*/");
      fout << token;
    }

    // Skip over quoted string.
    else if(token == "\"") {
      do {
        fout << token;
        gettoken(token);
      } while(token != "\"");
      fout << token;
    }

    else if(token == "foreach") foreach();

    else if(token == "cases") cases();

    else if(token == "repeat") repeat();

    else if(token == "until") until();

    else if(token == "typeof") typeof();

    else fout << token;
  }
} catch(SyntaxExc exc) {
  cout << exc.geterror() << endl;
  return 1;
}
```

每次循环都会从输入文件中读入一个令牌。如果这个令牌不需要转换，就将其写入到输出文件。然而，如果这个令牌包含了一个实验性关键字，就会调用合适的函数将这个关键字转换为对应的C++代码。注意，这个循环忽略了注释以及引号中的字符串，并将其复制到输出文件中，而没有检查它们的内容中是否包含实验性关键字。为了阻止对注释中的关键字或者引号中字符串中包含的关键字的转换，这样做是有必要的。

如果在试验性特征的转换中发生了语法错误，执行转换的代码会抛出 SyntaxExc 异常，这

个异常会被 main()中的 catch 捕获，它只是简单地显示这个错误。您可以增强这个错误报告，如果您需要的话，可以在其中包含行数或者其他信息。

4.5.3 gettoken()和 skipspaces()函数

为了将实验性的关键字转换为等价的 C++代码，译码器必须知道何时发现了一个关键字。为此，输入文件必须由一个令牌接一个令牌地处理。在此，术语标记是一个很松散的概念，它只是意味着一小段文本。译码器并不需要处理全部的令牌，它只需要认出标识符(包括关键字)、数字以及空白。大多数其他的字符可以作为单个字符来处理。

gettoken()函数如下所示：

```
// Get the next token from the input stream.
bool gettoken(string &tok) {
  char ch;
  char ch2;
  static bool trackIndent = true;

  tok = "";

  ch = fin.get();

  // Check for EOF and return false if EOF
  // is found.
  if(!fin) return false;

  // Read whitespace.
  if(isspace(ch)) {
    while(isspace(ch)) {
      tok += ch;

      // Reset indent counter with each new line.
      if(ch == '\n') {
        indent = "";
        trackIndent = true;
      }
      else if(trackIndent) indent += ch;

      ch = fin.get();
    }
    fin.putback(ch);
    return true;
  }

  // Stop tracking indentation after encountering
  // first non-whitespace character on a line.
  trackIndent = false;
  // Read an identifier or keyword.
  if(isalpha(ch) || ch=='_') {
    while(isalpha(ch) || isdigit(ch) || ch=='_') {
      tok += ch;
```

```
    ch = fin.get();
  }
  fin.putback(ch);
  return true;
}

// Read a number.
if(isdigit(ch)) {
  while(isdigit(ch) || ch=='.' ||
        tolower(ch) == 'e' ||
        ch == '-' || ch =='+') {
    tok += ch;
    ch = fin.get();
  }
  fin.putback(ch);
  return true;
}

// Check for \"
if(ch == '\\') {
  ch2 = fin.get();
  if(ch2 == '"') {
    tok += ch;
    tok += ch2;
    ch = fin.get();
  } else
    fin.putback(ch2);
}

// Check for '"'
if(ch == '\'') {
  ch2 = fin.get();
  if(ch2 == '"') {
    tok += ch;
    tok += ch2;
    return true;
  } else
    fin.putback(ch2);
}

// Check for begin comment symbols.
if(ch == '/') {
  tok += ch;
  ch = fin.get();
  if(ch == '/' || ch == '*') {
    tok += ch;
  }
  else fin.putback(ch);
  return true;
}

// Check for end comment symbols.
```

```
  if(ch == '*') {
    tok += ch;
    ch = fin.get();
    if(ch == '/') {
      tok += ch;
    }
    else fin.putback(ch);
    return true;
  }

  tok += ch;

  return true;
}
```

gettoken()函数有一个参数，一个名为 tok 的字符串，传递一个 string 对象的引用。当函数返回时，这个对象将包含令牌。如果某个令牌被读取，则函数返回 true；如果碰到文件结尾，则返回 false。

gettoken()函数开始将 tok 设置为 null 字符串。然后从 fin 中读取下一个字符，并将其存储在 ch 中。如果读取时碰到了文件结尾，则返回 false。否则，会测试 ch 的各种可能性。

首先，如果 ch 为空白，就会进入一个循环来读取空白字符，直到读取了第一个非空白字符为止。空白字符追加在 tok 后面。通过调用 fin.putback()将非空白字符返回到输入流中。在循环的结尾，tok 包含了所有被读取的空白字符，这就是返回到调用例程的标记。

在空白循环中还有其他一些事情需要注意。如果读取新的一行，indent 则被设置为 null 字符串，trackIndent 变量被设置为 true。当 trackIndent 为 true 时，空白将存储在 indent 中。在空白循环之后，trackIndent 被设置为 false。因此，在 indent 中只存储一行开头的空白。

如果 ch 不是空白字符，就会测试其他的可能性。如果下一个标记是标识符或者关键字，它以字母或者下划线开始，并且被 gettoken()中的下一个循环读取。否则，如果 ch 是一个数字，就会读取一个数值。

如果 ch 不是空白、字母、下划线或者数字，那么就会检查 5 种特殊的状态。第一个是\"序列。如前所述，译码器不会对引号内的文本执行转换。当译码器发现开引号时，只是简单地将它与闭引号之间的文本复制。然而，有必要阻止嵌套的引号被误认为是引号内字符串的开始或者结尾。因此，反斜杠序列\"必须被作为一对字符来处理。当某个字符常量指定一个引号时，情况相同。注释符号也必须作为一对字符来处理，因此 gettoken()检查//、/*，以及*/。

当执行转换时，有些情况下必须抛弃实验性语句中的空白，因为它们与生成的 C++代码没有关系。在此显示的 skipspaces()函数可以完成这个任务。它只是读取并舍弃空白。

```
void skipspaces() {
  char ch;

  do {
    ch = fin.get();
  } while(isspace(ch));
  fin.putback(ch);
}
```

4.5.4 转换 foreach 循环

最具有挑战性的是对 foreach 循环的转换。处理这个问题的函数 foreach()显示如下：

```
// Translate a foreach loop.
void foreach() {
  string token;
  string varname;
  string arrayname;

  char forvarname[5] = "_i";
  static char counter[2] = "a";

  // Create loop control variable for generated
  // for loop.
  strcat(forvarname, counter);
  counter[0]++;

  // Only 26 foreach loops in a file because
  // generated loop control variables limited to
  // _ia to _iz. This can be changed if desired.
  if(counter[0] > 'z')
    throw SyntaxExc("Too many foreach loops.");

  fout << "int " << forvarname
       << " = 0;\n";

  // Write beginning of generated for loop.
  fout << indent << "for(";

  skipspaces();

  // Read the (
  gettoken(token);
  if(token[0] != '(')
    throw SyntaxExc("( expected in foreach.");

  skipspaces();

  // Get the type of the foreach variable.
  gettoken(token);
  fout << token << " ";

  skipspaces();

  // Read and save the foreach variable's name.
  gettoken(token);
  varname = token;

  skipspaces();

  // Read the "in"
```

```
  gettoken(token);
  if(token != "in")
    throw SyntaxExc("in expected in foreach.");

  skipspaces();

  // Read the array name.
  gettoken(token);
  arrayname = token;

  fout << varname << " = " << arrayname << "[0];\n";

  // Construct target value.
  fout << indent + " " << forvarname << " < "
       << "((sizeof " << token << ")/"
       << "(sizeof " << token << "[0]));\n";

  fout << indent + " " << forvarname << "++, "
       << varname << " = " << arrayname << "["
       << forvarname << "])";

  skipspaces();

  // Read the )
  gettoken(token);
  if(token[0] != ')')
    throw SyntaxExc(") expected in foreach.");
}
```

这个译码器将 foreach 循环转换为等价的 for 循环。这涉及到两个相当复杂的问题的处理。为了理解它们，考虑下面的示例：

```
double nums[] = { 1.1, 2.2, 3.3 };
double sum = 0.0;

foreach(double v in nums)
  sum += v;
```

foreach 循环被转换为下面的 C++代码：

```
int _ia = 0;
for(double v = nums[0];
    _ia < ((sizeof nums)/(sizeof nums[0]));
    _ia++, v = nums[_ia])
  sum += v;
```

首先，注意 for 循环需要一个循环控制变量，在这里称之为_ia，但是在 foreach 语句中没有这种变量。记住，在 foreach 内声明的变量(在此情况下为 v)接收数组元素的值。它不是循环计数器。这意味着必须创建循环控制变量。这个变量不能在 for 循环内声明，因为这个声明必须为 foreach 变量 v 的声明保存，v 必须是循环内的局部变量。因此，必须在 for 循环外声明这个循环控制变量。从而要求生成一个惟一的整型变量名称，这个名称不与这个作用域内使用的任

何其他变量冲突。

其次，for 必须遍历数组中(在这里为 nums)的所有元素，但是数组的大小不是 foreach 说明的一部分。这意味着必须计算数组中元素的个数。幸运的是，可以使用 sizeof 完成此任务。

现在让我们浏览转换过程中的每一个步骤。foreach 转换的开始创建了一个(希望如此)惟一的变量，这个变量将用作 for 循环控制变量。这个变量的名称以_i 开始，随后是 a 到 z 之间的一个字母。在文件中创建的第一个变量的名称为_ia，第二个为_ib，等。总共有 26 个这样的变量。这并不是生成无冲突变量名称的最好方法，但是它的优点是简单，对于试验 foreach 循环来说已经足够了。这个变量的类型为 int，其声明写入到输出文件。

随后，向输出文件写入“for(”，for 循环由此开始。然后创建 for 循环的初始化部分。首先读取 foreach 变量的类型和名称，并将其复制到输出文件。随后，读取关键字 in，然后将其丢弃。再读取数组的名称，并用它来创建初始化部分，将 foreach 变量的值设置为数组第一个元素的值。

随后生成 for 循环的条件部分。在这个部分，循环控制变量与代表了数组元素数量的值比较。这个值是使用数组的大小除以数组中单个元素的大小来获得的。这个除法是必须的，因为 sizeof 返回数组的字节数，而不是数组中元素的数量。

最后，创建 for 循环的迭代部分。它增加循环控制变量，并且将 foreach 变量的值设置为数组下一个元素的值。在返回之前，foreach()确定 foreach 语句以“)”结尾。

最后一点：在 C#中，foreach 可以用于数组和集合(类似于 STL 容器)。为了简单起见，这个版本只能用于数组。然而，您可以试着修改它，使其能够用于遍历 STL 容器。

4.5.5 转换 cases 语句

如前所述，cases 语句提供了用一条语句取代一系列堆叠的 case 语句的方法。它是由这里给出的 cases()函数转换的：

```
// Translate a cases statement.
void cases() {
  string token;
  int start, end;

  skipspaces();

  // Get starting value.
  gettoken(token);

  if(isdigit(token[0])) {
    // is an int constant
    start = atoi(token.c_str());
  }
  else if(token[0] == '\'') {
    // is char constant
    gettoken(token);

    start = (int) token[0];

    // discard closing '
```

```
    gettoken(token);
    if(token[0] != '\'')
      throw SyntaxExc("' expected in cases.");
  }
  else
    throw SyntaxExc("Constant expected in cases.");

  skipspaces();

  // Read and discard the "to".
  gettoken(token);
  if(token != "to")
    throw SyntaxExc("to expected in cases.");

  skipspaces();

  // Get ending value.
  gettoken(token);

  if(isdigit(token[0])) {
    // is an int constant
    end = atoi(token.c_str());
  }
  else if(token[0] == '\'') {
    // is char constant
    gettoken(token);

    end = (int) token[0];

    // discard closing '
    gettoken(token);
    if(token[0] != '\'')
      throw SyntaxExc("' expected in cases.");
  }
  else
    throw SyntaxExc("Constant expected in cases.");

  skipspaces();

  // Read and discard the :
  gettoken(token);

  if(token != ":")
    throw SyntaxExc(": expected in cases.");

  // Generate stacked case statements.
  fout << "case " << start << ":\n";
  for(int i = start+1 ; i <= end; i++) {
    fout << indent << "case " << i << ":";
    if(i != end) fout << endl;
  }
}
```

尽管看起来代码很多，但实际上这只是一个简单的转换。cases 语句可以让您指定一个与 switch 表达式匹配的值的范围。译码器只是将这个范围转换为一系列堆叠的 case 语句。例如，这条 switch 语句：

```
switch(val) {
  cases 0 to 3:
    cout << "val is 0, 1, 2, or 3\n";
    break;
  case 4:
    cout << "val is 4\n";
}
```

被转换为等价的 C++代码：

```
switch(val) {
  case 0:
  case 1:
  case 2:
  case 3:
    cout << "val is 0, 1, 2, or 3\n";
    break;
  case 4:
    cout << "val is 4\n";
}
```

正如您看到的那样，在此生成了代表 cases 语句指定的值范围的单个 case 语句。

cases()函数是这样运行的。首先，读取这个范围的起始值。这个值或者是整数，或者是字符常量。如果是整型常量，则以数字开始。如果是字符常量，则以单引号开始。这个值作为整型值存储在 start 中。随后，读取并丢弃 to。然后读取结束值，它也是一个整型值或者字符常量。它的值存储在整型变量 end 中。最后，输出了一系列堆叠的 case 语句，每一条语句对应这个范围中的一个值。

4.5.6 转换 typeof 运算符

typeof 运算符支持另一种比较两种类型的方法。如前所述，其一般的形式为：

```
typeof op1 same as op2
```

在此，op1 和 op2 可以是标识符(如 int 或者 MyClass)或者对象。因此，typeof 可以用来比较两个对象的类型、一个对象的类型与已知类型，或者两个类型。如果两个类型相同，则返回 true；否则返回 false。typeof 运算符被转换为 C++表达式，来比较对每个操作数都应用了 typeid 运算符后得到的结果。

typeof 的转换由 typeof()函数来处理，如下所示：

```
// Translate a typeof expression.
void typeof() {
  string token;
  string temp;

  fout << "typeid(";
```

```
  skipspaces();

  gettoken(token);

  do {
    temp = token;

    if(!gettoken(token))
      throw SyntaxExc("Unexpected EOF encountered.");

    if(token != "same") fout << temp;
  } while(token != "same");

  skipspaces();

  gettoken(token);

  if(token != "as") throw SyntaxExc("as expected.");

  fout << ") == typeid(";

  skipspaces();

  do {
    if(!gettoken(token))
      throw SyntaxExc("Unexpected EOF encountered.");

    fout << token;
  } while(token != ")");
  fout << ")";
}
```

typeof()的操作很容易跟踪。它读取第一个操作数，并输出包含了这个类型的 typeid 表达式。然后读取 same 和 as 关键字，并将其舍弃。随后读取第二个操作数，并输出相应的 typeid 表达式。因此，typeof 语句：

```
if(typeof obj same as Myclass)
  cout << "obj is a MyClass object.\n";
```

被转换为下面的 C++代码：

```
if(typeid(obj) == typeid(Myclass))
  cout << "obj is a MyClass object.\n";
```

4.5.7 转换 repeat/until 循环

如前所述，用译码器处理 repeat/until 循环主要是为了说明问题，因为很容易使用宏处理这个问题。然而，让译码器执行这个转换确实节省了您的精力，使得您不需要在每个使用它们的程序中都定义 repeat 和 until 宏。它还可以作为一个模型，可以对其进行修改来适应其他类型的循环。

使用 repeat()方法转换关键字 repeat，如下所示，它简单地被替换为 do：

```
// Translate a repeat loop.
void repeat() {
  fout << "do";
}
```

关键字 until 由 until()来处理，如下所示：

```
// Translate an until.
void until() {
  string token;
  int parencount = 1;

  fout << "while";

  skipspaces();

  // Read and store the (
  gettoken(token);
  if(token != "(")
    throw SyntaxExc("( expected in typeof.");
  fout << "(";

  // Begin while by reversing and
  // parenthesizing the condition.
  fout << "!(";

  // Now, read the expression.
  do {
    if(!gettoken(token))
      throw SyntaxExc("Unexpected EOF encountered.");

    if(token == "(") parencount++;
    if(token == ")") parencount--;

    fout << token;
  } while(parencount > 0);
  fout << ")";
}
```

until()函数用关键字 while 取代 until，然后反转条件表达式。(记住，repeat/until 循环在条件为 true 时终止。do/while 循环在条件为 false 时终止)。

尽管在概念上很简单，但在译码器的这个部分反转条件表达式还是需要花费一点力气。原因是不能简单地在表达式的开始加“!”。另外，必须将“!”开头的表达式加上括号。为了理解这样做的原因，考虑下面的 repeat/until 循环：

```
int i=0;

repeat {
  cout << i << " ";
```

```
  i-+;
} until (i==10);
```

它被转换为如下的 do/while 循环：

```
int i=0;

do {
  cout << i << " ";
  i++;
} while(!(i==10));
```

while 条件表达式中的括号确保 i 不等于 10 的时候循环一直运行。然而，如果去掉括号，如下所示：

```
}while(!i==10);
```

循环在“!i”等于 10 时运行，这是一个完全不同的条件。

为了给 until 表达式加括号，译码器必须在表达式的开头加入左括号，在结尾加入右括号。问题是，译码器如何知道条件表达式何时结束呢？它通过对右括号计数做出判断。until 中的表达式包含在一对括号中。括号的计数存储在 parencount 变量中，这个变量的初始值为 1(对应于 until 表达式的左括号)。当复制条件表达式时，每发现一次左括号就增加这个计数，每发现一次右括号就减小这个计数。因此，当 parencount 为 0 时，就到达了表达式的结尾，就可以添加最后的右括号了。

4.6 演示程序

下面的程序演示了译码器支持的实验性功能。

```
// Demonstrate all of the experimental features
// supported by the translator.
#include <iostream>
using namespace std;

// Create a polymorphic base class.
class A {
public:
  virtual void f() { };
};

// And a concrete subclass.
class B: public A {
public:
  void f() {}
};

int main() {
  int n[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  double dn[] = {1.1, 2.2, 3.3, 4.4 };
```

```
cout << "Using a foreach loop.\n";

/* Keywords, such as foreach or typeof
   are ignored when inside comments
   or quoted strings. */

// A foreach loop.
foreach(int x in n )
  cout << x << ' ';

cout << "\n\n";

cout << "Using nested foreach loops.\n";

// A foreach loop with a block.
foreach(double f in dn) {
  cout << f << ' ';
  cout << f*f << ' ';

  // A nested foreach loop.
  foreach(double f in dn)
    cout << f/3 << " ";

  cout << endl;
}

cout << endl;

cout << "Demonstrate cases statement.\n";

cout << "A cases statement that uses integer constants:\n";

// Demonstrate cases statement that uses
// integer constants.
for(int i=0; i < 12; i++)
  switch(i) {
    case 0:
      cout << "case 0\n";
      break;
    cases 1 to 6:
      cout << "cases 1 to 6\n";
      break;
    case 7:
      cout << "case 7\n";
      break;
    cases 8 to 10:
      cout << "cases 8 to 10\n";
      break;
    default:
      cout << "case 11\n";
  }
```

```
  cout << "\n";

  cout << "A cases statement that uses character constants:\n";

  // Demonstrate a cases statement that uses
  // character constants.
  for(char ch='a'; ch <= 'e'; ch++)
    switch(ch) {
      case 'a':
        cout << "case a\n";
        break;
      cases 'b' to 'd':
        cout << "cases b to d\n";
        break;
      case 'e':
        cout << "case e\n";
    }

  cout << endl;

  cout << "A repeat/until loop.\n";

  // Demonstrate a repeat/until loop.
  int k = 0;
  repeat {
    k++;
    cout << "k: " << k << " ";
  } until(k==10);

  cout << "\n\n";

  cout << "Use typeof.\n";

  // Demonstrate typeof.
  A *aPtr;
  B *bPtr, bObj;

  // Assign a base pointer to derived object.
  aPtr = &bObj;
  bPtr = &bObj;

  if(typeof *aPtr same as *bPtr)
    cout << "aPtr points to same type of object as bPtr\n";

  if(typeof *aPtr same as B)
    cout << "aPtr points to B object\n";

  return 0;
}
```

当此程序通过译码器运行时，就生成了如下的C++代码：

```
// Translated from an .exp source file.
// Demonstrate all of the experimental features
// supported by the translator.
#include <iostream>
using namespace std;

// Create a polymorphic base class.
class A {
public:
  virtual void f() { };
};

// And a concrete subclass.
class B: public A {
public:
  void f() {}
};

int main() {
  int n[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  double dn[] = {1.1, 2.2, 3.3, 4.4 };

  cout << "Using a foreach loop.\n";

  /* Keywords, such as foreach or typeof
     are ignored when inside comments
     or quoted strings. */

  // A foreach loop.
  int _ia = 0;
  for(int x = n[0];
     _ia < ((sizeof n)/(sizeof n[0]));
     _ia++, x = n[_ia])
    cout << x << ' ';

  cout << "\n\n";

  cout << "Using nested foreach loops.\n";

  // A foreach loop with a block.
  int _ib = 0;
  for(double f = dn[0];
     _ib < ((sizeof dn)/(sizeof dn[0]));
     _ib++, f = dn[_ib]) {
    cout << f << ' ';
    cout << f*f << ' ';

    // A nested foreach loop.
    int _ic = 0;
    for(double f = dn[0];
       _ic < ((sizeof dn)/(sizeof dn[0]));
       _ic++, f = dn[_ic])
```

```
    cout << f/3 << " ";

  cout << endl;
}

cout << endl;

cout << "Demonstrate cases statement.\n";

cout << "A cases statement that uses integer constants:\n";

// Demonstrate cases statement that uses
// integer constants.
for(int i=0; i < 12; i++)
  switch(i) {
    case 0:
      cout << "case 0\n";
      break;
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
    case 6:
      cout << "cases 1 to 6\n";
      break;
    case 7:
      cout << "case 7\n";
      break;
    case 8:
    case 9:
    case 10:
      cout << "cases 8 to 10\n";
      break;
    default:
      cout << "case 11\n";
  }

cout << "\n";

cout << "A cases statement that uses character constants:\n";

// Demonstrate a cases statement that uses
// character constants.
for(char ch='a'; ch <= 'e'; ch++)
  switch(ch) {
    case 'a':
      cout << "case a\n";
      break;
    case 98:
    case 99:
    case 100:
```

```
      cout << "cases b to d\n";
      break;
    case 'e':
      cout << "case e\n";
  }

  cout << endl;

  cout << "A repeat/until loop.\n";

  // Demonstrate a repeat/until loop.
  int k = 0;
  do {
    k++;
    cout << "k: " << k << " ";
  } while(!(k==10));

  cout << "\n\n";

  cout << "Use typeof.\n";

  // Demonstrate typeof.
  A *aPtr;
  B *bPtr, bObj;

  // Assign a base pointer to derived object.
  aPtr = &bObj;
  bPtr = &bObj;

  if(typeid(*aPtr) == typeid(*bPtr))
    cout << "aPtr points to same type of object as bPtr\n";

  if(typeid(*aPtr) == typeid(B))
    cout << "aPtr points to B object\n";

  return 0;
}
```

这个C++版本可以被任何流行的C++编译器编译，并生成了如下的输出：

```
Using a foreach loop.
1 2 3 4 5 6 7 8 9 10

Using nested foreach loops.
1.1 1.21 0.366667 0.733333 1.1 1.46667
2.2 4.84 0.366667 0.733333 1.1 1.46667
3.3 10.89 0.366667 0.733333 1.1 1.46667
4.4 19.36 0.366667 0.733333 1.1 1.46667

Demonstrate cases statement.
A cases statement that uses integer constants:
case 0
```

```
cases 1 to 6
cases 1 to 6
cases 1 to 6
cases 1 to 6
cases 1 to 6
cases 1 to 6
case 7
cases 8 to 10
cases 8 to 10
cases 8 to 10
case 11

A cases statement that uses character constants:
case a
cases b to d
cases b to d
cases b to d
case e

A repeat/until loop.
k: 1 k: 2 k: 3 k: 4 k: 5 k: 6 k: 7 k: 8 k: 9 k: 10

Use typeof.
aPtr points to same type of object as bPtr
aPtr points to B object
```

4.7 尝试完成以下任务

正如本章开始所讲述的那样，译码器的目的是为了使您能够按照自己的想法实现新的语言特征或者有趣地改变现有特征。为了在译码器中加入您自己的实验性关键字，首先要创建一个函数来处理这个转换。然后，在主循环内加入另外的 else if 语句，当遇到新的关键字时，就会调用这个函数。

您可以试验任何您想到的结构。下面的想法可以帮助您开始：

- 一条 breakon 语句，当一些条件为 true 时打断一个循环。它可能使用这样的语法：

```
breakon(x==99);
```

- 一条 breakto 语句，将控制转移到循环之外，或者跳转到指定的标记。它可能使用这样的语法：

```
breakto jmp12;
```

- 一条 ignore 语句，当碰到某个值时，它引起循环的提前迭代。因此，它改进了 if...continue 语句。它可能使用这样的语法：

```
ignore(n==12);
```

当然，并不是所有的实验性功能都具有好的结果。

最后一点：如果您有兴趣设计自己的语言，就会发现第 9 章特别有价值，因为在那里为 C++的一个小子集开发了一个解释程序。您可以将这个解释程序作为起点，来寻求能够轻易处理比本章创建的译码器更为高级的语言特性。