第 9 章　Mini C++解释程序

前面的章节已经揭示了 C++语言的丰富与强大，并描述了可以应用它的广泛任务。本章作为本书的最后一章，将从不同的观点来审视 C++：设计的艺术。C++的强大功能是建立在优雅的、逻辑上一致的结构基础上的。正是这种坚实的架构将 C++的各个特性凝聚为一个有聚合力的整体。

为了探索 C++优雅的设计，本章开发了一个程序来理解这门语言。在此有两种类型的程序可以做到这一点：编译器和解释程序。尽管为 C++创建一个编译器远远超出了本书的范围，但创建一个解释程序却是比较容易管理的任务。因此，本章为 C++的一个子集开发了一个解释程序。这个解释程序称为 Mini C++。由于 Mini C++解释 C++语言，因此它的实现用具体的示例说明了 C++语法的基本原理。

除了演示 C++逻辑上一致的本性之外，Mini C++还有另外一个优点：它给予您一个可执行的引擎，您可以按需要增强或者修改它。例如，可以扩展 Mini C++来解释一个更大的 C++子集，或者解释一种完全不同的计算机语言。甚至可以将它作为测试平台来试验您自己设计的计算机语言。它的使用是不受限制的。

9.1　解释程序和编译器

在开始之前，有必要解释一下什么是解释程序，以及它与编译器的区别。尽管编译器和解释程序都是将程序的源代码作为输入，但是它们处理源代码的方式却有很大的区别。

编译器将程序的源代码转换为一种可以执行的形式。通常，例如在 C++中，这种可执行的形式由 CPU 指令组成，CPU 指令可以直接由计算机执行。在其他情况下，编译器的输出是一种可移植的中间形式，这种形式可以被运行系统执行。例如，Java 会编译一种称为字节码的中间形式，字节码由 Java 虚拟机执行。

解释程序以完全不同的方式运行。它将源代码读入程序，并执行碰到的每一条语句。因此，解释程序不会将源代码转换为可执行代码。相反，解释程序直接执行这个程序。解释方式的主要缺点是降低了执行速度。以解释方式运行的程序要比编译形式慢很多。

有时术语“解释程序”会用在不同于刚才描述的情况下。例如，最初的 Java 虚拟机被称为“字节码解释程序”。这与本章所开发的解释程序不是一种类型。在此开发的解释程序是“源代码”解释程序，意味着它通过简单地阅读源代码来执行程序。

尽管编译的程序比解释的程序执行快，但是解释程序在程序设计中的应用仍然很普遍，原因如下。首先，解释程序提供了一个真正的交互式环境，其中程序可以由用户的交互来暂停或者重新启动。举例来说，这种交互式环境很适合于机器人技术。其次，由于语言解释程序的本性，它们特别适合于交互式调试。第三，解释程序对于“脚本语言”(如数据库的查询语言)很优秀。第四，它们允许在不同的平台上运行相同的程序。每个新的环境只需要实现解释程序的

运行包。

还有一个原因使得解释程序很有趣：它们易于修改、变更或者增强。这意味着如果您想要创建、试验或者控制您自己的语言，那么使用解释程序要比使用编译器容易。解释程序创建了一个了不起的语言原型环境，因为您可以改变语言，并很快看到结果。

当然，对于本章的目的而言，解释程序的最大好处是使得您能够直接看到C++的内部运行。

9.2 Mini C++纵览

Mini C++由许多代码组成。在开始之前，大体上理解Mini C++的组织方式是有好处的。Mini C++包含了3个主要的子系统：处理数学表达式的表达式解析器；实际执行程序的解释程序；还有一组库函数。

表达式解析器求解数学表达式的值，并返回结果。表达式可以包含常量，如10或者88，变量、函数调用，当然还包含运算符。例如：

```
x+num/32
```

是一个表达式。

解释程序模块执行C++程序。解释程序通过每次读入源代码的一个令牌来运行。当它遇到关键字时，会满足关键字的任何请求。例如，当解释程序读取if时，就会处理if指定的条件。如果这个条件为true，则解释程序执行与if相关的代码块。解释程序一直运行，直到程序结束。

库子系统定义了Mini C++中内建的函数。因此，它们组成了Mini C++的“标准库”。在本章只包含了少数的库函数，但是只要您愿意，就可以加入任意数量的其他函数。总的来说，Mini C++的库函数与真正的C++库函数的目的相同。

Mini C++将这些子系统组织到3个文件中，如表9-1所示。

表9-1 子系统组织到文件中

子 系 统	文 件 名
解析器	parser.cpp
解释程序	minicpp.cpp
库	libcpp.cpp

Mini C++还使用了头文件mccommon.h。

9.3 Mini C++说明

尽管C++的关键字相对比较少，但它仍然是一门丰富而复杂的语言。描述并实现一个针对整个C++的解释程序远远超出了本章的范围。取而代之的是，Mini C++只能理解这门语言相当有限的子集。然而，这个特定的子集包含了许多C++最重要的特性。例如，Mini C++支持递归函数、全局和局部变量、嵌套的作用域以及大多数的控制语句。Mini C++支持的所有特性的列表如下所示：

- 具有局部变量的参数化函数

- 嵌套的作用域
- 递归
- if 语句
- switch 语句
- do-while、while 以及 for 循环
- break 语句
- int、char 类型的局部和全局变量
- 整型和字符型的函数参数
- 整型和字符型常量
- 字符串常量(部分实现)
- return 语句，可以带返回值，也可以不带返回值。
- 少量的标准库函数
- 运算符+、－、*、/、%、<、>、<=、>=、==、!=、++、－－、一元－以及一元+
- 返回整型数的函数
- “/*”和“//”注释
- 使用 cin 和 cout 的控制台 I/O

虽然这个列表看起来很短，但是实现它却需要许多代码。原因之一是当解释像 C++这样的语言时，必须支付许多的“入场费用”。尽管刚才列出的特性只是 C++语言的一个小子集，但它们确实使得这个解释程序能够处理这门语言的核心，包括基础语法、控制语句、表达式以及函数调用机制。因此，Mini C++处理了语言的“骨干”。

毫无疑问，您注意到了 Mini C++不支持类。这样做的原因只是考虑到实际情况。支持类意味着解释程序也支持用户自定义类型、对象实例化以及点(.)运算符。另外，类要求解释程序理解 public 和 private。尽管解释程序处理这些特性本身并不困难，然而解释程序的代码将会比本章所显示的大的多。一旦您理解了解释程序的运行方式，您或许会发现亲自加入对类的支持很有趣。

另一些没有支持的特性包括函数以及运算符的重载、模板、命名空间、异常、预处理器、结构、联合以及位运算。同样，解释这些特性并不困难，但是这样做会使得代码不可预知地增长，从而不能以书本的形式来使用。然而，加入对这些特性的支持确实是一个有趣的项目，您或许想亲自体验它。

Mini C++的一些限制

虽然只实现了 C++的一个小子集，但是 Mini C++的源代码相当长。为了阻止它变得更长，对 C++程序员强加了一些限制。首先，if、while、do 以及 for 的目标必须是用左括号和右括号包围起来的代码块。不能使用一条单个的语句。例如，Mini C++将不会正确地解释如下的代码：

```
for(a=0; a < 10; a++)
  for(b=0; b < 10; b++)
    for(c=0; c < 10; c++)
      cout << "hi";

if(...)
  if(...) x = 10;
```

取而代之的是，您必须以这种方式来编写代码：

```
for(a=0; a < 10; a++) {
  for(b=0; b < 10; b++) {
    for(c=0; c < 10; c++) {
      cout << "hi";
    }
  }
}

if(...) {
  if(...) {
    x = 10;;
  }
}
```

这些限制使得解释程序能够比较容易地发现构成这些程序控制语句目标的代码结尾。然而，由于这些程序控制语句的对象经常是代码块，因此这些限制并不是很苛刻。(如果您愿意的话，只需要一点点功夫，就可以去除这些限制)。

另一个限制是不支持原型。所有的函数都被假定返回一个整型值(也允许返回字符型，但是会转化为整型)，并且不会执行参数类型的检查。另外，由于不支持函数重载，因此函数必须有且仅有一个版本。

除了不支持 default 语句外，switch 语句完全实现。这个限制是为了降低解释 switch 语句所需代码的大小和复杂程度。(switch 语句是解释起来很复杂的语句之一)。对 default 的支持需要您自己来尝试。

最后，所有的函数前面都必须使用 int 或者 char 类型说明符。因此，Mini C++解释程序不支持 void 返回类型。从而，下面的声明是有效的：

```
int f()
{
  // ...
}
```

但是这个声明是无效的：

```
void f()
{
  // ...
}
```

9.4 非正式的 C++理论

为了充分理解 Mini C++的操作，有必要理解 C ++语言的构造方式。如果您已经看到过 C++语言的正式说明(如 ANSI/ISO C++标准中的内容)，就知道这些说明非常长，并且充满了相当晦涩的语句。不要着急——您不需要正式地学习 C++语言的说明来理解 Mini C++。Mini C++解释的那部分 C++很容易理解。也就是说，您对 C++语言的定义方式只需要有基本的了解。对于我们的目的而言，下面非正式的讨论就足够了。然而要记住，这些讨论有意简化了一些概念。

C++程序由一个或者多个函数的集合以及全局变量(如果有的话)组成。函数由函数名、参数列表以及函数体组成，函数体是一个代码块。代码块以“{”开始，随后是一条或者多条语句，并且以}结束。代码块(也称为复合语句)创建了一个作用域。通常，语句或者是表达式、嵌套的代码块，或者以关键字(如 if、for 或者 int)开始。嵌套的代码块创建了嵌套的作用域。

提示：

尽管前面对语句(关键字、代码块以及表达式)的分类对于理解 Mini C++已经足够，但是 C++的 ANSI/ISO 标准使用了更细粒度的定义，它将语句定义为可标记的(labeled)、表达式、复合、选择、迭代、跳转、声明以及 try 块。

C++程序通过调用 main()开始运行。当在 main()中遇到最后的“}”或者 return 时，程序结束——假定在其他地方没有调用 exit()或者 abort()。程序中的其他函数都必须直接或者间接地被 main()函数调用。因此，当Mini C++运行一个程序时，只是简单地从main()函数开始并且在main()函数结束时停止。

9.4.1 C++表达式

C++的基础是表达式，因为在程序中的许多操作都是通过表达式发生的。为了理解原因，回忆一下，C++有三类主要的语句：关键字、语句块以及表达式。这意味着按照定义，任何不是基于关键字的语句或者没有定义语句块的语句都是表达式语句。因此，下面的语句都是表达式：

```
count = 100; // line 1
sample = i / 22 * (c-10); // line 2
num = abs(count) * 2; // line 3
strcpy(str1, str2); // line 4
```

让我们仔细观察每一条表达式语句。在 C++中，等号是一个赋值运算符。这很重要。C++没有像某些语言那样，把赋值当作单独的语句类型。相反，等号是一种赋值运算符，赋值操作产生的值等于右边表达式产生的值。因此在 C++中，一条赋值语句实际上是一个赋值表达式。由于它是表达式，因此具有值。这就是下面的表达式合法的原因：

```
a = b = c = 100;
if( (a=4+5) == 0 ) ...;
```

这些表达式可以运行的原因是赋值是一个可以产生值的操作。

第 2 行显示了一个更加复杂的赋值表达式。在此情况下，计算右边表达式的值，并赋给 sample。

在第 3 行，调用了 abs()来返回参数的绝对值。因此，表达式中使用函数会导致调用这个函数。

正如第 4 行所示的那样，函数调用本身就是表达式。在此情况下，调用了 strcpy()来将一个字符串的内容复制到另一个字符串。虽然 strcpy()的返回值被忽略，但是 strcpy()的调用仍然构成一个表达式。

9.4.2 定义表达式

为了对表达式求值，Mini C++使用了一个表达式解析器。在理解表达式解析器的操作之前，

需要了解表达式的构造方法。在几乎所有的计算机语言中，都是用一组产生式规则(production rule)来描述表达式的，产生式规则定义了操作数与运算符的交互。定义了 Mini C++表达式的产生式规则如图 9-1 所示。

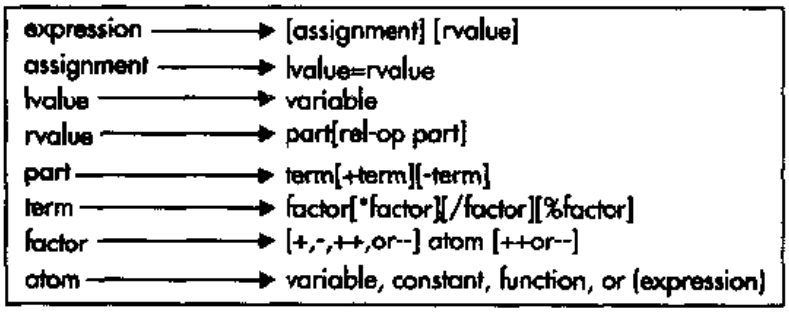

图 9-1　Mini C++表达式的产生式规则

在此，rel-op 指任何的 C++关系运算符。术语 lvalue 和 rvalue 分别指赋值语句左边和右边的对象。箭头指“产生”。有一点需要知道，运算符的优先级内建于产生式规则中。优先级越高，运算符在列表中的位置就越靠下。正如产生式规则所显示的那样，Mini C++支持如下的运算符：

```
+      -      *      /      %      ++      --
<      <=     >      >=     ==     !=      ( )
```

为了观察这些规则的运行方式，对如下的 C++表达式求值：

```
count=10-5*3;
```

首先，应用规则 1，将这个表达式分解为下面 3 个部分。见图 9-2。

图 9-2　表达式的 3 个部分

由于在分解后表达式的“rvalue”部分没有关系运算符，因此应用了术语产生式规则。参见图 9-3。

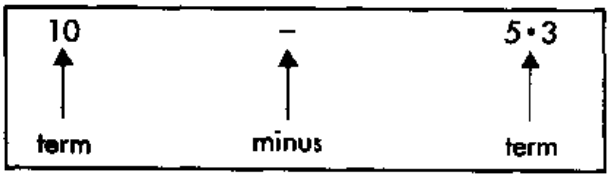

图 9-3　产生式规则

当然，第 2 项由因子 3 和 5 组成。这两个因子是常量，代表了产生式规则的最底层。随后，再次回到这个规则来计算表达式的值。首先，5*3 等于 15。然后，用 10 减这个值，得到-5。最后，这个值赋给 count，这也是整个表达式的值。Mini C++表达式解析器以相同的方式求解这个表达式。它简单地实现了产生式规则。

由于表达式解析器对 Mini C++非常重要，因此下面我们来详细介绍表达式解析器。

9.5 表达式解析器

读取并分析表达式的代码段称为表达式解析器。毫无疑问，表达式解析器是 Mini C++需要的最重要的子系统。由于 C++定义的表达式比许多其他语言广泛的多，因此表达式解析器执行了组成 C++程序的大量代码。

可以通过多种不同的方法来为 C++设计一个表达式解析器。许多商业编译器使用了表格驱动的解析器，这个解析器通常由解析器生成器程序创建。尽管表格驱动的解析器通常比其他方法要快，但是它们很难手工创建。Mini C++使用了递归下降解析器(recursive-descent parser)，这个解析器在逻辑上实现了刚才讨论的产生式规则。

递归下降解析器本质上是相互递归函数的集合，此集合处理表达式。如果这个解析器在编译器中使用，那么会生成对应于源代码的正确的对象代码。然而在解释程序中，解析器求解给定表达式的值。在这个部分，开发了 Mini C++解析器。

提示：

作者编写范围广泛的表达式解析器已经很多年了。当涉及到 C++时，对这个问题的深入观察请参考作者的 C++书籍：The Complete Reference，McGraw-Hill/Osborne。

9.5.1 解析器代码

Mini C++递归下降解析器的整个代码如下所示。这些代码保存在文件 parser.cpp 中。后面部分将解释这个解析器的操作。

```
// Recursive descent parser for integer expressions.
//
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <cctype>
#include "mccommon.h"

using namespace std;

// Keyword lookup table.
// Keywords must be entered lowercase.
struct commands {
  char command[20];
  token_ireps tok;
} com_table[] = {
  "if", IF,
  "else", ELSE,
  "for", FOR,
  "do", DO,
  "while", WHILE,
  "char", CHAR,
  "int", INT,
  "return", RETURN,
  "switch", SWITCH,
```

```
  "break", BREAK,
  "case", CASE,
  "cout", COUT,
  "cin", CIN,
  "", END // mark end of table
};

// This structure links a library function name
// with a pointer to that function.
struct intern_func_type {
  char *f_name; // function name
  int (*p)(); // pointer to the function
} intern_func[] = {
  "getchar", call_getchar,
  "putchar", call_putchar,
  "abs", call_abs,
  "rand", call_rand,
  "", 0 // null terminate the list
};

// Entry point into parser.
void eval_exp(int &value)
{
  get_token();

  if(!*token) {
    throw InterpExc(NO_EXP);
  }

  if(*token == ';') {
    value = 0; // empty expression
    return;
  }
  eval_exp0(value);

  putback(); // return last token read to input stream
}

// Process an assignment expression.
void eval_exp0(int &value)
{
  // temp holds name of var receiving the assignment.
  char temp[MAX_ID_LEN+1];

  tok_types temp_tok;

  if(token_type == IDENTIFIER) {
    if(is_var(token)) { // if a var, see if assignment
      strcpy(temp, token);
      temp_tok = token_type;
      get_token();
      if(*token == '=') { // is an assignment
```

```
      get_token();
      eval_exp0(value); // get value to assign
      assign_var(temp, value); // assign the value
      return;
    }
    else { // not an assignment
      putback(); // restore original token
      strcpy(token, temp);
      token_type = temp_tok;
    }
  }
 }
 eval_exp1(value);
}

// Process relational operators.
void eval_exp1(int &value)
{
 int partial_value;
 char op;
 char relops[] = {
  LT, LE, GT, GE, EQ, NE, 0
 };

 eval_exp2(value);
 op = *token;
 if(strchr(relops, op)) {
  get_token();
  eval_exp2(partial_value);

  switch(op) { // perform the relational operation
    case LT:
      value = value < partial_value;
      break;
    case LE:
      value = value <= partial_value;
      break;
    case GT:
      value = value > partial_value;
      break;
    case GE:
      value = value >= partial_value;
      break;
    case EQ:
      value = value == partial_value;
      break;
    case NE:
      value = value != partial_value;
      break;
  }
 }
}
```

```
// Add or subtract two terms.
void eval_exp2(int &value)
{
  char op;
  int partial_value;
  char okops[] = {
    '(', INC, DEC, '-', '+', 0
  };

  eval_exp3(value);

  while((op = *token) == '+' || op == '-') {
    get_token();

    if(token_type == DELIMITER &&
       !strchr(okops, *token))
         throw InterpExc(SYNTAX);
      eval_exp3(partial_value);

      switch(op) { // add or subtract
        case '-':
          value = value - partial_value;
          break;
        case '+':
          value = value + partial_value;
          break;
    }
  }
}

// Multiply or divide two factors.
void eval_exp3(int &value)
{
  char op;
  int partial_value, t;
  char okops[] = {
    '(', INC, DEC, '-', '+', 0
  };

  eval_exp4(value);

  while((op = *token) == '*' || op == '/'
        || op == '%') {
    get_token();

    if(token_type == DELIMITER &&
       !strchr(okops, *token))
         throw InterpExc(SYNTAX);

    eval_exp4(partial_value);
```

```
    switch(op) { // mul, div, or modulus
      case '*':
        value = value * partial_value;
        break;
      case '/':
        if(partial_value == 0)
          throw InterpExc(DIV_BY_ZERO);
        value = (value) / partial_value;
        break;
      case '%':
        t = (value) / partial_value;
        value = value - (t * partial_value);
        break;
    }
  }
}

// Is a unary +, -, ++, or --.
void eval_exp4(int &value)
{
  char op;
  char temp;

  op = '\0';
  if(*token == '+' || *token == '-' ||
     *token == INC || *token == DEC)
  {
    temp = *token;
    op = *token;
    get_token();
    if(temp == INC)
      assign_var(token, find_var(token)+1);
    if(temp == DEC)
      assign_var(token, find_var(token)-1);
  }

  eval_exp5(value);
  if(op == '-') value = -(value);
}

// Process parenthesized expression.
void eval_exp5(int &value)
{

  if((*token == '(')) {
    get_token();

    eval_exp0(value); // get subexpression

    if(*token != ')')
      throw InterpExc(PAREN_EXPECTED);
    get_token();
```

```
  }
  else
    atom(value);
}

// Find value of number, variable, or function.
void atom(int &value)
{
  int i;
  char temp[MAX_ID_LEN+1];

  switch(token_type) {
    case IDENTIFIER:
      i = internal_func(token);
      if(i != -1) {
        // Call "standard library" function.
        value = (*intern_func[i].p)();
      }
      else if(find_func(token)) {
        // Call programmer-created function.
        call();
        value = ret_value;
      }
      else {
        value = find_var(token); // get var's value
        strcpy(temp, token); // save variable name

        // Check for ++ or --.
        get_token();
        if(*token == INC || *token == DEC) {
          if(*token == INC)
            assign_var(temp, find_var(temp)+1);
          else
            assign_var(temp, find_var(temp)-1);
        } else putback();
      }

      get_token();
      return;
    case NUMBER: // is numeric constant
      value = atoi(token);
      get_token();

      return;
    case DELIMITER: // see if character constant
      if(*token == '\'') {
        value = *prog;
        prog++;
        if(*prog!='\'')
          throw InterpExc(QUOTE_EXPECTED);
        prog++;
        get_token();
```

```
      return ;
    }
    if(*token==')') return; // process empty expression
    else throw InterpExc(SYNTAX); // otherwise, syntax error
  default:
    throw InterpExc(SYNTAX); // syntax error
 }
}

// Display an error message.
void sntx_err(error_msg error)
{
 char *p, *temp;
 int linecount = 0;

 static char *e[]= {
  "Syntax error",
  "No expression present",
  "Not a variable",
  "Duplicate variable name",
  "Duplicate function name",
  "Semicolon expected",
  "Unbalanced braces",
  "Function undefined",
  "Type specifier expected",
  "Return without call",
  "Parentheses expected",
  "While expected",
  "Closing quote expected",
  "Division by zero",
  "{ expected (control statements must use blocks)",
  "Colon expected"
 };

 // Display error and line number.
 cout << "\n" << e[error];
 p = p_buf;
 while(p != prog) { // find line number of error
  p++;
  if(*p == '\r') {
   linecount++;
  }
 }
 cout << " in line " << linecount << endl;

 temp = p;
 while(p > p_buf && *p != '\n') p--;

 // Display offending line.
 while(p <= temp)
  cout << *p++;
```

```
  cout << endl;
}

// Get a token.
tok_types get_token()
{

  char *temp;

  token_type = UNDEFTT; tok = UNDEFTOK;

  temp = token;
  *temp = '\0';

  // Skip over white space.
  while(isspace(*prog) && *prog) ++prog;

  // Skip over newline.
  while(*prog == '\r') {
    ++prog;
    ++prog;
    // Again, skip over white space.
    while(isspace(*prog) && *prog) ++prog;
  }

  // Check for end of program.
  if(*prog == '\0') {
    *token = '\0';
    tok = END;
    return (token_type = DELIMITER);
  }

  // Check for block delimiters.
  if(strchr("{}", *prog)) {
    *temp = *prog;
    temp++;
    *temp = '\0';
    prog++;
    return (token_type = BLOCK);
  }

  // Look for comments.
  if(*prog == '/')
    if(*(prog+1) == '*') { // is a /* comment
      prog += 2;
      do { // find end of comment
        while(*prog != '*') prog++;
        prog++;
      } while (*prog != '/');
      prog++;
      return (token_type = DELIMITER);
```

```
  } else if(*(prog+1) == '/') { // is a // comment
    prog += 2;
    // Find end of comment.
    while(*prog != '\r' && *prog != '\0') prog++;
    if(*prog == '\r') prog +=2;
    return (token_type = DELIMITER);
  }

// Check for double-ops.
if(strchr("!<>=+-", *prog)) {
  switch(*prog) {
    case '=':
      if(*(prog+1) == '=') {
        prog++; prog++;
        *temp = EQ;
        temp++; *temp = EQ; temp++;
        *temp = '\0';
      }
      break;
    case '!':
      if(*(prog+1) == '=') {
        prog++; prog++;
        *temp = NE;
        temp++; *temp = NE; temp++;
        *temp = '\0';
      }
      break;
    case '<':
      if(*(prog+1) == '=') {
      prog++; prog++;
      *temp = LE; temp++; *temp = LE;
    }
    else if(*(prog+1) == '<') {
      prog++; prog++;
      *temp = LS; temp++; *temp = LS;
    }
    else {
      prog++;
      *temp = LT;
    }
    temp++;
    *temp = '\0';
    break;
  case '>':
    if(*(prog+1) == '=') {
      prog++; prog++;
      *temp = GE; temp++; *temp = GE;
    } else if(*(prog+1) == '>') {
      prog++; prog++;
      *temp = RS; temp++; *temp = RS;
    }
    else {
```

```
      prog++;
      *temp = GT;
    }
    temp++;
    *temp = '\0';
    break;
  case '+':
    if(*(prog+1) == '+') {
      prog++; prog++;
      *temp = INC; temp++; *temp = INC;
      temp++;
      *temp = '\0';
    }
    break;
  case '-':
    if(*(prog+1) == '-') {
      prog++; prog++;
      *temp = DEC; temp++; *temp = DEC;
      temp++;
      *temp = '\0';
    }
    break;
  }

  if(*token) return(token_type = DELIMITER);
}

// Check for other delimiters.
if(strchr("+-*^/%=;:(),'", *prog)) {
  *temp = *prog;
  prog++;
  temp++;
  *temp = '\0';
  return (token_type = DELIMITER);
}

// Read a quoted string.
if(*prog == '"') {
  prog++;
  while(*prog != '"' && *prog != '\r' && *prog) {
    // Check for \n escape sequence.
    if(*prog == '\\') {
      if(*(prog+1) == 'n') {
        prog++;
        *temp++ = '\n';
      }
    }
    else if((temp - token) < MAX_T_LEN)
      *temp++ = *prog;

    prog++;
  }
```

```
  if(*prog == '\r' || *prog == 0)
    throw InterpExc(SYNTAX);
  prog++; *temp = '\0';
  return (token_type = STRING);
 }

 // Read an integer number.
 if(isdigit(*prog)) {
  while(!isdelim(*prog)) {
    if((temp - token) < MAX_ID_LEN)
      *temp++ = *prog;
    prog++;
  }
  *temp = '\0';
  return (token_type = NUMBER);
 }

 // Read identifier or keyword.
 if(isalpha(*prog)) {
  while(!isdelim(*prog)) {
    if((temp - token) < MAX_ID_LEN)
      *temp++ = *prog;
    prog++;
  }
  token_type = TEMP;
 }

 *temp = '\0';

 // Determine if token is a keyword or identifier.
 if(token_type == TEMP) {
  tok = look_up(token); // convert to internal form
  if(tok) token_type = KEYWORD; // is a keyword
  else token_type = IDENTIFIER;
 }

 // Check for unidentified character in file.
 if(token_type == UNDEFTT)
   throw InterpExc(SYNTAX);

 return token_type;
}

// Return a token to input stream.
void putback()
{
 char *t;

 t = token;
 for(; *t; t++) prog--;
}
```

```
// Look up a token's internal representation in the
// token table.
token_ireps look_up(char *s)
{
  int i;
  char *p;

  // Convert to lowercase.
  p = s;
  while(*p) { *p = tolower(*p); p++; }

  // See if token is in table.
  for(i=0; *com_table[i].command; i++) {
    if(!strcmp(com_table[i].command, s))
      return com_table[i].tok;
  }

  return UNDEFTOK; // unknown command
}

// Return index of internal library function or -1 if
// not found.
int internal_func(char *s)
{
  int i;

  for(i=0; intern_func[i].f_name[0]; i++) {
    if(!strcmp(intern_func[i].f_name, s)) return i;
  }
  return -1;
}

// Return true if c is a delimiter.
bool isdelim(char c)
{
  if(strchr(" !:;,+-<>'/*%^=()", c) || c == 9 ||
     c == '\r' || c == 0) return true;
  return false;
}
```

这个解析器使用了下面的全局变量和枚举类型(包含在本章后面显示的头文件 mccommon.h 中):

```
const int MAX_T_LEN = 128; // max token length
const int MAX_ID_LEN = 31; // max identifier length

// Enumeration of token types.
enum tok_types { UNDEFTT, DELIMITER, IDENTIFIER,
                 NUMBER, KEYWORD, TEMP, STRING, BLOCK };

// Enumeration of internal representation of tokens.
enum token_ireps { UNDEFTOK, ARG, CHAR, INT, SWITCH,
```

```
                    CASE, IF, ELSE, FOR, DO, WHILE, BREAK,
                    RETURN, COUT, CIN, END };

// Enumeration of two-character operators, such as <=.
enum double_ops { LT=1, LE, GT, GE, EQ, NE, LS, RS, INC, DEC };

// These are the constants used when throwing a
// syntax error exception.
//
// NOTE: SYNTAX is a generic error message used when
// nothing else seems appropriate.
enum error_msg
     { SYNTAX, NO_EXP, NOT_VAR, DUP_VAR, DUP_FUNC,
       SEMI_EXPECTED, UNBAL_BRACES, FUNC_UNDEF,
       TYPE_EXPECTED, RET_NOCALL, PAREN_EXPECTED,
       WHILE_EXPECTED, QUOTE_EXPECTED, DIV_BY_ZERO,
       BRACE_EXPECTED, COLON_EXPECTED };

extern char *prog; // current location in source code
extern char *p_buf; // points to start of program buffer

extern char token[MAX_T_LEN+1]; // string version of token
extern tok_types token_type; // contains type of token
extern token_ireps tok; // internal representation of token

extern int ret_value; // function return value

// Exception class for Mini C++.
class InterpExc {
  error_msg err;
public:
  InterpExc(error_msg e) { err = e; }
  error_msg get_err() { return err; }
};
```

prog 指向源代码中当前正在执行的位置。因此，prog 保存了一个地址，解释程序将在这个位置读取下一块程序。解释程序不会改变 p_buf 指针，这个指针总是指向被解释程序的开始。当前的令牌存储在 token 中。(令牌是程序代码中不可分割的部分。下面部分将对其进行更精确的定义)。令牌的类型存储在 token_type 中。如果这个令牌是关键字，那么 tok 变量保存这个令牌的内部形式。

枚举类型 tok_types 声明了 Mini C++能够识别的令牌的类型。token_ireps 枚举类型指定了代表关键字的令牌的内部格式。代表由两个字符组成的运算符(例如<=)的值由 double_ops 枚举类型指定。error.msg 枚举了各种错误代码。最后，InterPExc 是用来报告语法错误的异常类。

9.5.2 分解源代码

将源代码分解为组成部件的机制是所有解释程序(以及编译器，在此情况下)的基础，这些组成部件称为令牌。令牌是程序不可分割的部分，代表一个逻辑单元。例如，{、+、==、以及 if 都是令牌。虽然等号运算符“==”由两个字符组成，但是它们不能被分割，否则就会改变符号

含义。因此，==是一个独立的逻辑单元，从而是一个令牌。同样，if 也是一个令牌，但是 i 或者 f 对于 C++都没有特殊的含义。

C++的 ANSI/ISO 标准定义了下面类型的令牌：

关键字　　标识符　　字面值
运算符　　标点

关键字是组成 C++语言的那些令牌，例如 while。标识符是变量、函数等的名称。字面值包含字符串和常量值，如数字 10。运算符含义自明。标点包括分号、逗号、花括号和圆括号。(某些标点符号也是运算符，取决于它们的使用)。

尽管 Mini C++可以识别的令牌与 ANSI/ISO C++标准指定的令牌相同，但是识别的方式却有很大的区别，目的是使得解释容易一点。Mini C++使用的令牌分类如表 9-2 所示。

表 9-2　Mini C++使用的令牌分类

令牌类型	包含内容
分隔符	标点和运算符
关键字	关键字
字符串	引用字符串
标识符	变量和函数名称
数值	数值常量
代码块	{or}

让我们完成一个示例，来说明这些令牌类型的应用方式。给定下面的语句：

```
for(x=0; x<10; x++) {
  num = num + x;
}
```

生成了如表 9-3 所示的令牌。

表 9-3　生成的令牌

令　牌	类　别
for	关键字
(	分隔符
x	标识符
=	分隔符
0	数值
;	分隔符
x	标识符
<	分隔符
10	数值
;	分隔符
x	标识符

（续表）

令　牌	类　别
++	分隔符
)	分隔符
{	代码块
num	标识符
=	分隔符
num	标识符
+	分隔符
x	标识符
;	分隔符
}	代码块

get_token()函数为 Mini C++解释程序从源代码中返回令牌代码，如下所示：

```
// Get a token.
tok_types get_token()
{

  char *temp;

  token_type = UNDEFTT; tok = UNDEFTOK;

  temp = token;
  *temp = '\0';

  // Skip over white space.
  while(isspace(*prog) && *prog) ++prog;

  // Skip over newline.
  while(*prog == '\r') {
    ++prog;
    ++prog;
    // Again, skip over white space.
    while(isspace(*prog) && *prog) ++prog;
  }

// Check for end of program.
if(*prog == '\0') {
  *token = '\0';
  tok = END;
  return (token_type = DELIMITER);
}

// Check for block delimiters.
if(strchr("{}", *prog)) {
```

```
  *temp = *prog;
  temp++;
  *temp = '\0';
  prog++;
  return (token_type = BLOCK);
}

// Look for comments.
if(*prog == '/')
  if(*(prog+1) == '*') { // is a /* comment
    prog += 2;
    do { // find end of comment
      while(*prog != '*') prog++;
      prog++;
    } while (*prog != '/');
    prog++;
    return (token_type = DELIMITER);
  } else if(*(prog+1) == '/') { // is a // comment
    prog += 2;
    // Find end of comment.
    while(*prog != '\r' && *prog != '\0') prog++;
    if(*prog == '\r') prog +=2;
    return (token_type = DELIMITER);
  }

// Check for double-ops.
if(strchr("!<>=+-", *prog)) {
  switch(*prog) {
    case '=':
      if(*(prog+1) == '=') {
        prog++; prog++;
        *temp = EQ;
        temp++; *temp = EQ; temp++;
        *temp = '\0';
      }
      break;
    case '!':
      if(*(prog+1) == '=') {
        prog++; prog++;
        *temp = NE;
        temp++; *temp = NE; temp++;
        *temp = '\0';
      }
      break;
    case '<':
      if(*(prog+1) == '=') {
        prog++; prog++;
        *temp = LE; temp++; *temp = LE;
      }
      else if(*(prog+1) == '<') {
        prog++; prog++;
        *temp = LS; temp++; *temp = LS;
```

```
    }
    else {
      prog++;
      *temp = LT;
    }
    temp++;
    *temp = '\0';
    break;
  case '>':
    if(*(prog+1) == '=') {
      prog++; prog++;
      *temp = GE; temp++; *temp = GE;
    } else if(*(prog+1) == '>') {
      prog++; prog++;
      *temp = RS; temp++; *temp = RS;
    }
    else {
      prog++;
      *temp = GT;
    }
    temp++;
    *temp = '\0';
    break;
  case '+':
    if(*(prog+1) == '+') {
      prog++; prog++;
      *temp = INC; temp++; *temp = INC;
      temp++;
      *temp = '\0';
    }
    break;
  case '-':
    if(*(prog+1) == '-') {
      prog++; prog++;
      *temp = DEC; temp++; *temp = DEC;
      temp++;
      *temp = '\0';
    }
    break;
 }

 if(*token) return(token_type = DELIMITER);
}

// Check for other delimiters.
if(strchr("+-*^/%=;:(),'", *prog)) {
  *temp = *prog;
  prog++;
  temp++;
  *temp = '\0';
  return (token_type = DELIMITER);
}
```

```
// Read a quoted string.
if(*prog == '"') {
  prog++;
  while(*prog != '"' && *prog != '\r' && *prog) {
    // Check for \n escape sequence.
    if(*prog == '\\') {
      if(*(prog+1) == 'n') {
        prog++;
        *temp++ = '\n';
      }
    }
    else if((temp - token) < MAX_T_LEN)
      *temp++ = *prog;
      prog++;
    }
    if(*prog == '\r' || *prog == 0)
      throw InterpExc(SYNTAX);
    prog++; *temp = '\0';
    return (token_type = STRING);
  }

  // Read an integer number.
  if(isdigit(*prog)) {
    while(!isdelim(*prog)) {
      if((temp - token) < MAX_ID_LEN)
        *temp++ = *prog;
      prog++;
    }
    *temp = '\0';
    return (token_type = NUMBER);
  }

  // Read identifier or keyword.
  if(isalpha(*prog)) {
    while(!isdelim(*prog)) {
      if((temp - token) < MAX_ID_LEN)
        *temp++ = *prog;
      prog++;
    }
    token_type = TEMP;
  }

  *temp = '\0';

  // Determine if token is a keyword or identifier.
  if(token_type == TEMP) {
    tok = look_up(token); // convert to internal form
    if(tok) token_type = KEYWORD; // is a keyword
    else token_type = IDENTIFIER;
  }
```

```
  // Check for unidentified character in file.
  if(token_type == UNDEFTT)
    throw InterpExc(SYNTAX);

  return token_type;
}
```

get_token()函数开始忽略所有的空白，包括回车和换行。由于令牌不会包含空格(除非是被引用的字符串或者字符常量)，因此必须忽略空格。get_token()函数还忽略了所有的注释。然后读取程序中的下一个令牌。注意各种令牌的处理方式。例如，如果程序中的下一个字符是数字，就会读取一个数字。如果下一个字符是字母，就会获取一个标识符或者关键字等。

令牌的字符串表示被放入 token 中。在读取时，令牌的类型(tok_types 枚举所枚举的那些)被放入 token_type，如果令牌是一个关键字，则它的内部表示(例如 token_ireps 枚举的)通过 look_up()函数赋给 tok。(后面将解释关键字内部表示的原因)。正如您看到的那样，get_token()将两个字符的 C++运算符，如<=或者++，转换为对应的枚举值。尽管技术上并没有这样要求，但是这样做会使得解析器易于实现。

9.5.3 显示语法错误

如果解析器碰到语法错误，就会抛出 InterpExc 异常，并指定与所发现的错误类型对应的枚举值。(Mini C++的其他部分还用 InterpExc 来报告错误)。InterpExc 异常的处理程序在 main()函数中，这是解释程序的主要文件 minicpp.cpp 的一部分，将在后面描述。这个处理程序调用了如下所示的 sntx_err()函数来报告这个错误：

```
// Display an error message.
void sntx_err(error_msg error)
{
  char *p, *temp;
  int linecount = 0;

  static char *e[]= {
    "Syntax error",
    "No expression present",
    "Not a variable",
    "Duplicate variable name",
    "Duplicate function name",
    "Semicolon expected",
    "Unbalanced braces",
    "Function undefined",
    "Type specifier expected",
    "Return without call",
    "Parentheses expected",
    "While expected",
    "Closing quote expected",
    "Division by zero",
    "{ expected (control statements must use blocks)",
    "Colon expected"
  };
```

```
  // Display error and line number.
  cout << "\n" << e[error];
  p = p_buf;
  while(p != prog) { // find line number of error
    p++;
    if(*p == '\r') {
      linecount++;
    }
  }
  cout << " in line " << linecount << endl;

  temp = p;
  while(p > p_buf && *p != '\n') p--;

  // Display offending line.
  while(p <= temp)
    cout << *p++;

  cout << endl;
}
```

注意，sntx_err()显示了一个字符串来描述错误以及发现错误的那一行的编号(可能是发生错误那行的下一行)。这个函数还显示了发现错误的那一行。

9.5.4 表达式求值

以 eval_exp 开头的函数和 atom()函数实现了 Mini C++表达式的产生式规则。为了理解解析器对表达式求值的方式，对下面的表达式进行操作。(假定 prog 指向表达式的开始)。

```
10-3*2
```

当调用解析器的入口函数 eval_exp()时，首先获取第一个令牌。如果这个令牌为空，就会抛出一个异常，指示没有出现表达式。然而，在此情况下，令牌包含了数字 10。由于令牌不为空，因此调用 exval_exp0()，这个函数检查赋值运算符。由于没有出现赋值，因此调用 eval_exp1()，这个函数依次调用了 eval_exp2()。接着，eval_exp2()调用 eval_exp3()，eval_exp3()调用 eval_exp4()。eval_exp4()处理一元+和 -，以及前缀++和- -。这个函数随后调用 eval_exp5()。在 eval_exp5()函数中，或者递归调用 eval_exp0()(在使用了括号的表达式中)，或者递归调用 atom()找到一个值。由于这个令牌不是左括号，因此执行 atom()，给 value 赋值 10。随后检索另一个令牌，函数沿着链条向上返回。现在这个令牌是运算符 -，函数一直返回到 eval_exp2()。

接下来发生的事情很重要。由于这个令牌是 -，因此将其保存在 op 中。然后解析器获取下一个令牌，这个令牌是 3，这个链条重新向下开始。与前面一样，进入 atom()。3 返回到 value 中，读取运算符*。从而导致链条返回到 eval_exp3()，在此处读取最后的令牌 2。在这里进行了第一个算术操作——2 乘以 3。这个结果返回到 eval_exp2()，并执行减运算。减运算产生了答案 4。尽管这个过程看起来很复杂，但您亲自来完成其他的示例就会清楚解析器的操作。

现在，让我们更加仔细地观察 atom()函数。这个函数寻找一个整型常量或者变量、函数或者字符常量的值。(这个函数还处理后缀++以及--)。在源代码中，可能出现两种类型的函数：用户自定义函数或者库函数。如果遇到了用户自定义函数，解释程序执行它的代码来判断返回

值。(在后面的部分将讨论 Mini C++实际完成函数调用的方式)。如果这个函数是库函数，首先使用函数 internal_func()来查找其地址，然后通过接口函数来访问它。库函数和接口函数的地址保存在 intern_func 数组中，如下所示：

```
// This structure links a library function name
// with a pointer to that function.
struct intern_func_type {
  char *f_name; // function name
  int (*p)(); // pointer to the function
} intern_func[] = {
  "getchar", call_getchar,
  "putchar", call_putchar,
  "abs", call_abs,
  "rand", call_rand,
  "", 0 // null terminate the list
};
```

正如您所看到的那样，Mini C++只解释少数的库函数，但是如果需要的话，很容易加入其他的函数。(实际的接口函数保存在一个单独的文件中，将在“Mini C++库函数”那一部分讨论)。

在表达式解析器文件中，关于这个例程的最后一点是：为了正确地分析 C++语法，有时需要“提前查找一个令牌”。例如，考虑下面的语句

```
alpha=count();
```

为了让 Mini C++知道 count 是一个函数而不是变量，必须读取 count 和下一个令牌。如果下一个令牌是括号，就像这里的情况，那么解析器知道 count 是一个函数。然而，如果这条语句是：

```
alpha = count*10;
```

那么 count 之后的下一个令牌是*。由于它不是圆括号，则意味着 count 一定是一个变量。如果没有提前读取下一个令牌，则解析器没有办法作出判断。在两种情况下，都需要把提前读取的令牌返回到输入流，以进行后面的处理。为此，表达式解析器文件包含了 putback()函数，这个函数返回从输入流读取的最后一个令牌。

9.6 Mini C++解释程序

实际执行 C++程序的引擎是解释程序。在进入代码之前，理解解释程序一般的操作方式是有好处的。在许多方面，解释程序的代码比表达式解析器的代码容易理解，因为在概念上，解释 C++程序的动作可以总结为以下的算法：

```
当(令牌存在){
  获取下一个令牌;
  采取正确的动作;
}
```

与表达式解析器相比，这个算法不可思议地简单，但是所有的解释程序确实都是这么做的。

需要记住的一点是，“采取正确的动作”步骤可能涉及到从输入流而来的其他令牌，并且需要进一步的正确动作。因此，“采取正确的动作”步骤可以是递归的。

为了理解这个算法确切的运行方式，让我们手工解释下面的代码段：

```
int a;

a = 10 * num;

if(a < 100) {
  cout << a;
}
```

按照这个算法，读取第一个令牌 int。对这个令牌执行的恰当动作是读取下一个令牌，以找出被声明变量的名称(在此情况下是 a)，然后存储这个名称。下一个令牌是结束这一行的分号。在此采取的正确动作是忽略它。现在，获取下一个令牌 a。由于这条语句不是以关键字开始，那么它一定是一个表达式。在此，正确的动作是使用解析器求这个表达式的值。这个过程耗尽了这条语句所有的令牌。随后读取了 if 令牌。这表示 if 语句的开始，因此正确的动作是处理 if，这意味着求条件表达式的值。如果表达式的结果为 true，就会解释与 if 语句相关的代码块，否则就会将其忽略。

任何 C++程序都会发生刚才描述的过程。直到读取了最后一个令牌或者遇到了语法错误，解释过程才会停止。记住基本的算法，让我们检查这个解释程序。整个的解释程序文件名为 **minicpp.cpp**，如下所示。随后的部分将详细分析这些代码。

```
// A Mini C++ interpreter.

#include <iostream>
#include <fstream>
#include <new>
#include <stack>
#include <vector>
#include <cstring>
#include <cstdlib>
#include <cctype>
#include "mccommon.h"

using namespace std;

char *prog; // current execution point in source code
char *p_buf; // points to start of program buffer

// This structure encapsulates the info
// associated with variables.
struct var_type {
  char var_name[MAX_ID_LEN+1]; // name
  token_ireps v_type; // data type
  int value; // value
};

// This vector holds info for global variables.
vector<var_type> global_vars;
```

```
// This vector holds info for local variables
// and parameters.
vector<var_type> local_var_stack;

// This structure encapsulates function info.
struct func_type {
  char func_name[MAX_ID_LEN+1]; // name
  token_ireps ret_type; // return type
  char *loc; // location of entry point in program
};

// This vector holds info about functions.
vector<func_type> func_table;

// Stack for managing function scope.
stack<int> func_call_stack;

// Stack for managing nested scopes.
stack<int> nest_scope_stack;

char token[MAX_T_LEN+1]; // current token
tok_types token_type; // token type
token_ireps tok; // internal representation

int ret_value; // function return value

bool breakfound = false; // true if break encountered

int main(int argc, char *argv[])
{
  if(argc != 2) {
    cout << "Usage: minicpp <filename>\n";
    return 1;
  }

  // Allocate memory for the program.
  try {
    p_buf = new char[PROG_SIZE];
  } catch (bad_alloc exc) {
    cout << "Could Not Allocate Program Buffer\n";
    return 1;
  }

  // Load the program to execute.
  if(!load_program(p_buf, argv[1])) return 1;

  // Set program pointer to start of program buffer.
  prog = p_buf;

  try {
    // Find the location of all functions
```

```
    // and global variables in the program.
    prescan();

    // Next, set up the call to main().

    // Find program starting point.
    prog = find_func("main");

    // Check for incorrect or missing main() function.
    if(!prog) {
      cout << "main() Not Found\n";
      return 1;
    }

    // Back up to opening (.
    prog--;

    // Set the first token to main
    strcpy(token, "main");

    // Call main() to start interpreting.
    call();
  }
  catch(InterpExc exc) {
    sntx_err(exc.get_err());
    return 1;
  }
  catch(bad_alloc exc) {
    cout << "Out Of Memory\n";
    return 1;
  }

  return ret_value;
}

// Load a program.
bool load_program(char *p, char *fname)
{
  int i=0;

  ifstream in(fname, ios::in | ios::binary);
  if(!in) {
    cout << "Cannot Open file.\n";
    return false;
  }

  do {
    *p = in.get();
    p++; i++;
  } while(!in.eof() && i < PROG_SIZE);
    if(i == PROG_SIZE) {
      cout << "Program Too Big\n";
```

```
    return false;
  }

  // Null terminate the program. Skip any EOF
  // mark if present in the file.
  if(*(p-2) == 0x1a) *(p-2) = '\0';
  else *(p-1) = '\0';

  in.close();

  return true;
}

// Find the location of all functions in the program
// and store global variables.
void prescan()
{
  char *p, *tp;
  char temp[MAX_ID_LEN+1];
  token_ireps datatype;
  func_type ft;

  // When brace is 0, the current source position
  // is outside of any function.
  int brace = 0;

  p = prog;

  do {
    // Bypass code inside functions
    while(brace) {
      get_token();
      if(tok == END) throw InterpExc(UNBAL_BRACES);
      if(*token == '{') brace++;
      if(*token == '}') brace--;
    }

    tp = prog; // save current position
    get_token();

    // See if global var type or function return type.
    if(tok==CHAR || tok==INT) {
      datatype = tok; // save data type
      get_token();

      if(token_type == IDENTIFIER) {
        strcpy(temp, token);
        get_token();

      if(*token != '(') { // must be global var
      prog = tp; // return to start of declaration
      decl_global();
```

```
      }
        else if(*token == '(') { // must be a function

          // See if function already defined.
          for(unsigned i=0; i < func_table.size(); i++)
            if(!strcmp(func_table[i].func_name, temp))
              throw InterpExc(DUP_FUNC);

          ft.loc = prog;
          ft.ret_type = datatype;
          strcpy(ft.func_name, temp);
          func_table.push_back(ft);

          do {
            get_token();
          } while(*token != ')');
          // Next token will now be opening curly
          // brace of function.
        }
        else putback();
      }
    }
    else {
      if(*token == '{') brace++;
      if(*token == '}') brace--;
    }
  } while(tok != END);
  if(brace) throw InterpExc(UNBAL_BRACES);
  prog = p;
}

// Interpret a single statement or block of code. When
// interp() returns from its initial call, the final
// brace (or a return) in main() has been encountered.
void interp()
{
  int value;
  int block = 0;

  do {
    // Don't interpret until break is handled.
    if(breakfound) return;

    token_type = get_token();

    // See what kind of token is up.
    if(token_type == IDENTIFIER ||
       *token == INC || *token == DEC)
    {
      // Not a keyword, so process expression.
      putback(); // restore token to input stream for
                 // further processing by eval_exp()
```

```
  eval_exp(value); // process the expression
  if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}
else if(token_type==BLOCK) { // block delimiter?
  if(*token == '{') { // is a block
    block = 1; // interpreting block, not statement
    // Record nested scope.
    nest_scope_stack.push(local_var_stack.size());
  }
  else { // is a }, so reset scope and return
    // Reset nested scope.
    local_var_stack.resize(nest_scope_stack.top());
    nest_scope_stack.pop();
    return;
  }
}
else // is keyword
  switch(tok) {
    case CHAR:
    case INT: // declare local variables
      putback();
      decl_local();
      break;
    case RETURN: // return from function call
      func_ret();
      return;
    case IF: // process an if statement
      exec_if();
      break;
    case ELSE: // process an else statement
      find_eob(); // find end of else block
                  // and continue execution
      break;
    case WHILE: // process a while loop
      exec_while();
      break;
    case DO: // process a do-while loop
      exec_do();
      break;
    case FOR: // process a for loop
      exec_for();
      break;
    case BREAK: // handle break
      breakfound = true;

      // Reset nested scope.
      local_var_stack.resize(nest_scope_stack.top());
      nest_scope_stack.pop();
      return;
    case SWITCH: // handle a switch statement
      exec_switch();
      break;
```

```
      case COUT: // handle console output
        exec_cout();
        break;
      case CIN: // handle console input
        exec_cin();
        break;
      case END:
        exit(0);
    }
  } while (tok != END && block);
  return;
}

// Return the entry point of the specified function.
// Return NULL if not found.
char *find_func(char *name)
{
  unsigned i;

  for(i=0; i < func_table.size(); i++)
    if(!strcmp(name, func_table[i].func_name))
      return func_table[i].loc;
      return NULL;
    }

    // Declare a global variable.
    void decl_global()
    {
      token_ireps vartype;
      var_type vt;

      get_token(); // get type

      vartype = tok; // save var type

      // Process comma-separated list.
      do {
        vt.v_type = vartype;
        vt.value = 0; // init to 0
        get_token(); // get name

        // See if variable is a duplicate.
        for(unsigned i=0; i < global_vars.size(); i++)
          if(!strcmp(global_vars[i].var_name, token))
            throw InterpExc(DUP_VAR);

        strcpy(vt.var_name, token);
        global_vars.push_back(vt);

        get_token();
      } while(*token == ',');
```

```
    if(*token != ';') throw InterpExc(SEMI_EXPECTED);
  }

  // Declare a local variable.
  void decl_local()
  {
    var_type vt;

    get_token(); // get var type
    vt.v_type = tok; // store type

    vt.value = 0; // init var to 0

    // Process comma-separated list.
    do {
    get_token(); // get var name

    // See if variable is already the name
    // of a local variable in this scope.
    if(!local_var_stack.empty())
    for(int i=local_var_stack.size()-1;
        i >= nest_scope_stack.top(); i--)
    {
      if(!strcmp(local_var_stack[i].var_name, token))
        throw InterpExc(DUP_VAR);
    }

    strcpy(vt.var_name, token);
    local_var_stack.push_back(vt);
    get_token();
  } while(*token == ',');

  if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}

// Call a function.
void call()
{
  char *loc, *temp;
  int lvartemp;

  // First, find entry point of function.
  loc = find_func(token);

  if(loc == NULL)
    throw InterpExc(FUNC_UNDEF); // function not defined
  else {
    // Save local var stack index.
    lvartemp = local_var_stack.size();

    get_args(); // get function arguments
    temp = prog; // save return location
```

```
    func_call_stack.push(lvartemp); // push local var index

    prog = loc; // reset prog to start of function
    get_params(); // load the function's parameters with
                  // the values of the arguments

    interp(); // interpret the function
    prog = temp; // reset the program pointer

    if(func_call_stack.empty()) throw InterpExc(RET_NOCALL);

    // Reset local_var_stack to its previous state.
    local_var_stack.resize(func_call_stack.top());
    func_call_stack.pop();
  }
}

// Push the arguments to a function onto the local
// variable stack.
void get_args()
{
  int value, count, temp[NUM_PARAMS];
  var_type vt;

  count = 0;
  get_token();
  if(*token != '(') throw InterpExc(PAREN_EXPECTED);

  // Process a comma-separated list of values.
  do {
    eval_exp(value);
    temp[count] = value; // save temporarily
    get_token();
    count++;
  } while(*token == ',');
  count--;

  // Now, push on local_var_stack in reverse order.
  for(; count>=0; count--) {
    vt.value = temp[count];
    vt.v_type = ARG;
    local_var_stack.push_back(vt);
  }
}

// Get function parameters.
void get_params()
{
  var_type *p;
  int i;
```

```
  i = local_var_stack.size()-1;

  // Process comma-separated list of parameters.
  do {
    get_token();
    p = &local_var_stack[i];
    if(*token != ')' ) {
      if(tok != INT && tok != CHAR)
        throw InterpExc(TYPE_EXPECTED);

      p->v_type = tok;
      get_token();

      // Link parameter name with argument already on
      // local var stack.
      strcpy(p->var_name, token);
      get_token();
      i--;
    }
    else break;
  } while(*token == ',');

  if(*token != ')') throw InterpExc(PAREN_EXPECTED);
}

// Return from a function.
void func_ret()
{
  int value;

  value = 0;

  // Get return value, if any.
  eval_exp(value);

  ret_value = value;
}

// Assign a value to a variable.
void assign_var(char *vname, int value)
{
  // First, see if it's a local variable.
  if(!local_var_stack.empty())
    for(int i=local_var_stack.size()-1;
        i >= func_call_stack.top(); i--)
    {
      if(!strcmp(local_var_stack[i].var_name,
         vname))
      {
      if(local_var_stack[i].v_type == CHAR)
        local_var_stack[i].value = (char) value;
      else if(local_var_stack[i].v_type == INT)
```

```
        local_var_stack[i].value = value;
        return;
      }
    }

    // Otherwise, try global vars.
    for(unsigned i=0; i < global_vars.size(); i++)
      if(!strcmp(global_vars[i].var_name, vname)) {
        if(global_vars[i].v_type == CHAR)
          global_vars[i].value = (char) value;
        else if(global_vars[i].v_type == INT)
          global_vars[i].value = value;
        return;
      }

  throw InterpExc(NOT_VAR); // variable not found
}

// Find the value of a variable.
int find_var(char *vname)
{
  // First, see if it's a local variable.
  if(!local_var_stack.empty())
    for(int i=local_var_stack.size()-1;
        i >= func_call_stack.top(); i--)
    {
      if(!strcmp(local_var_stack[i].var_name, vname))
        return local_var_stack[i].value;
    }

    // Otherwise, try global vars.
    for(unsigned i=0; i < global_vars.size(); i++)
      if(!strcmp(global_vars[i].var_name, vname))
        return global_vars[i].value;

    throw InterpExc(NOT_VAR); // variable not found
  }

  // Execute an if statement.
  void exec_if()
  {
    int cond;

    eval_exp(cond); // get if expression.

    if(cond) { // if true, process target of IF
      // Confirm start of block.
      if(*token != '{')
        throw InterpExc(BRACE_EXPECTED);

      interp();
    }
```

```
  else {
    // Otherwise skip around IF block and
    // process the ELSE, if present.

    find_eob(); // find start of next line
    get_token();

    if(tok != ELSE) {
      // Restore token if no ELSE is present.
      putback();
      return;
    }

    // Confirm start of block.
    get_token();
    if(*token != '{')
      throw InterpExc(BRACE_EXPECTED);
    putback();

    interp();
  }
}

// Execute a switch statement.
void exec_switch()
{
  int sval, cval;
  int brace;

  eval_exp(sval); // Get switch expression.

  // Check for start of block.
  if(*token != '{')
    throw InterpExc(BRACE_EXPECTED);

  // Record new scope.
  nest_scope_stack.push(local_var_stack.size());

  // Now, check case statements.
  for(;;) {
    brace = 1;
    // Find a case statement.
    do {
      get_token();
      if(*token == '{') brace++;
      else if(*token == '}') brace--;
    } while(tok != CASE && tok != END && brace);

    // If no matching case found, then skip.
    if(!brace) break;

    if(tok == END) throw InterpExc(SYNTAX);
```

```
    // Get value of the case statement.
    eval_exp(cval);

    // Read and discard the :
    get_token();

    if(*token != ':')
      throw InterpExc(COLON_EXPECTED);

    // If values match, then interpret.
    if(cval == sval) {
      brace = 1;
      do {
        interp();

        if(*token == '{') brace++;
        else if(*token == '}') brace--;
      } while(!breakfound && tok != END && brace);

      // Find end of switch statement.
      while(brace) {
        get_token();
        if(*token == '{') brace++;
        else if(*token == '}') brace--;
      }
      breakfound = false;

      break;
    }
  }
}

// Execute a while loop.
void exec_while()
{
  int cond;
  char *temp;

  putback(); // put back the while
  temp = prog; // save location of top of while loop

  get_token();
  eval_exp(cond); // check the conditional expression

  // Confirm start of block.
  if(*token != '{')
    throw InterpExc(BRACE_EXPECTED);

  if(cond)
    interp(); // if true, interpret
  else { // otherwise, skip to end of loop
```

```
    find_eob();
    return;
  }

  prog = temp; // loop back to top

  // Check for break in loop.
  if(breakfound) {
    // Find start of loop block.
    do {
      get_token();
    } while(*token != '{' && tok != END );

    putback();
    breakfound = false;
    find_eob(); // now, find end of loop
    return;
  }
}

// Execute a do loop.
void exec_do()
{
  int cond;
  char *temp;


  // Save location of top of do loop.
  putback(); // put back do
  temp = prog;

  get_token(); // get start of loop block

  // Confirm start of block.
  get_token();
  if(*token != '{')
    throw InterpExc(BRACE_EXPECTED);
  putback();

  interp(); // interpret loop

  // Check for break in loop.
  if(breakfound) {
    prog = temp;
    // Find start of loop block.
    do {
      get_token();
    } while(*token != '{' && tok != END);

    // Find end of while block.
    putback();
    find_eob();
```

```
    // Now, find end of while expression.
    do {
      get_token();
    } while(*token != ';' && tok != END);
    if(tok == END) throw InterpExc(SYNTAX);

    breakfound = false;
    return;
  }

  get_token();
  if(tok != WHILE) throw InterpExc(WHILE_EXPECTED);
  eval_exp(cond); // check the loop condition

  // If true loop; otherwise, continue on.
  if(cond) prog = temp;
}

// Execute a for loop.
void exec_for()
{
  int cond;
  char *temp, *temp2;
  int paren ;

  get_token(); // skip opening (
  eval_exp(cond); // initialization expression

  if(*token != ';') throw InterpExc(SEMI_EXPECTED);
  prog++; // get past the ;
  temp = prog;

  for(;;) {
    // Get the value of the conditional expression.
    eval_exp(cond);

    if(*token != ';') throw InterpExc(SEMI_EXPECTED);
    prog++; // get past the ;
    temp2 = prog;

    // Find start of for block.
    paren = 1;
    while(paren) {
      get_token();
      if(*token == '(') paren++;
      if(*token == ')') paren--;
    }

    // Confirm start of block.
    get_token();
    if(*token != '{')
```

```
      throw InterpExc(BRACE_EXPECTED);
    putback();

    // If condition is true, interpret
    if(cond)
      interp();
    else { // otherwise, skip to end of loop
      find_eob();
      return;
    }

    prog = temp2; // go to increment expression

    // Check for break in loop.
    if(breakfound) {
      // Find start of loop block.
      do {
        get_token();
      } while(*token != '{' && tok != END);

      putback();
      breakfound = false;
      find_eob(); // now, find end of loop
      return;
    }

    // Evaluate the increment expression.
    eval_exp(cond);

    prog = temp; // loop back to top
  }
}

// Execute a cout statement.
void exec_cout()
{
  int val;

  get_token();
  if(*token != LS) throw InterpExc(SYNTAX);

  do {
    get_token();

    if(token_type==STRING) {
      // Output a string.
      cout << token;
    }
    else if(token_type == NUMBER ||
            token_type == IDENTIFIER) {
      // Output a number.
      putback();
```

```
      eval_exp(val);
      cout << val;
    }
    else if(*token == '\'') {
      // Output a character constant.
      putback();
      eval_exp(val);
      cout << (char) val;
    }

    get_token();
  } while(*token == LS);

  if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}

// Execute a cin statement.
void exec_cin()
{
  int val;
  char chval;
  token_ireps vtype;

  get_token();
  if(*token != RS) throw InterpExc(SYNTAX);

  do {
    get_token();
    if(token_type != IDENTIFIER)
      throw InterpExc(NOT_VAR);

    vtype = find_var_type(token);

    if(vtype == CHAR) {
      cin >> chval;
      assign_var(token, chval);
    }
    else if(vtype == INT) {
      cin >> val;
      assign_var(token, val);
    }

    get_token();
  } while(*token == RS);
  if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}

// Find the end of a block.
void find_eob()
{
  int brace;
```

```
  get_token();
  if(*token != '{' )
    throw InterpExc(BRACE_EXPECTED);

  brace = 1;

  do {
    get_token();
    if(*token == '{') brace++;
    else if(*token == '}') brace--;
  } while(brace && tok != END);

  if(tok==END) throw InterpExc(UNBAL_BRACES);
}

// Determine if an identifier is a variable. Return
// true if variable is found; false otherwise.
bool is_var(char *vname)
{
  // See if vname is a local variable.
  if(!local_var_stack.empty())
    for(int i=local_var_stack.size()-1;
        i >= func_call_stack.top(); i--)
    {
      if(!strcmp(local_var_stack[i].var_name, vname))
        return true;
    }

  // See if vname is a global variable.
  for(unsigned i=0; i < global_vars.size(); i++)
    if(!strcmp(global_vars[i].var_name, vname))
      return true;

  return false;
}

// Return the type of variable.
token_ireps find_var_type(char *vname)
{
  // First, see if it's a local variable.
  if(!local_var_stack.empty())
    for(int i=local_var_stack.size()-1;
        i >= func_call_stack.top(); i--)
    {
      if(!strcmp(local_var_stack[i].var_name, vname))
        return local_var_stack[i].v_type;
    }

// Otherwise, try global vars.
for(unsigned i=0; i < global_vars.size(); i++)
  if(!strcmp(global_vars[i].var_name, vname))
      return local_var_stack[i].v_type;
```

```
  return UNDEFTOK;
}
```

9.6.1 main()函数

main()函数开始解释您在命令行中指定的程序。如下所示：

```
int main(int argc, char *argv[])
{
  if(argc != 2) {
    cout << "Usage: minicpp <filename>\n";
    return 1;
  }

  // Allocate memory for the program.
  try {
    p_buf = new char[PROG_SIZE];
  } catch (bad_alloc exc) {
    cout << "Could Not Allocate Program Buffer\n";
    return 1;
  }

  // Load the program to execute.
  if(!load_program(p_buf, argv[1])) return 1;

  // Set program pointer to start of program buffer.
  prog = p_buf;
  try {
    // Find the location of all functions
    // and global variables in the program.
    prescan();

    // Next, set up the call to main().

    // Find program starting point.
    prog = find_func("main");

    // Check for incorrect or missing main() function.
    if(!prog) {
      cout << "main() Not Found\n";
      return 1;
    }

    // Back up to opening (.
    prog--;

    // Set the first token to main
    strcpy(token, "main");

    // Call main() to start interpreting.
    call();
```

```
  }

  catch(InterpExc exc) {
    sntx_err(exc.get_err());
    return 1;
  }
  catch(bad_alloc exc) {
    cout << "Out Of Memory\n";
    return 1;
  }

  return ret_value;
}
```

main()函数开始分配内存以保存被解释的程序。注意，可以被解释的最大程序由常量PROG_SIZE指定。这个值被任意地设置为10000，但是如果您愿意，可以增加它。随后，调用函数load_program()函数加载这个程序。在这个程序被加载之后，main()函数执行了3个主要的动作：

(1) 调用解释程序的预扫描函数prescan()。

(2) 查找main()在程序中的位置，使得解释程序准备好调用main()。

(3) 执行call()函数，在main()的起始位置开始执行这个程序。

main()函数还处理由Mini C++生成的InterPExc异常。这包括解析器抛出的那些异常。

下面部分详细讨论解释程序的关键组成部分。

9.6.2 解释程序的预扫描程序

解释程序在开始执行程序之前，必须执行两个重要的记录任务：

(1) 必须找到并初始化所有的全局变量。

(2) 必须发现程序中每个函数定义的位置。

这两个任务由解释程序的预扫描程序完成。

在Mini C++中，所有可执行的代码都存在于函数内部，因此一旦开始执行，Mini C++的解释程序没有理由到函数的外面。然而，全局变量的声明语句在所有函数的外面。因此，有必要使用预扫描程序来处理这些声明。解释程序没有其他(有效的)方法来了解它们。

为了保证执行的速度，需要知道每个函数在程序中定义的位置(尽管在技术上并不需要)，从而加快函数的调用。如果没有执行这个步骤，就需要搜索冗长的源代码，以找到每次调用函数的入口点。

寻找所有函数的入口点还有另一个目的。您知道，C++程序不是从这个程序的顶部开始执行的。相反，它是从main()函数开始的。另外，并没有要求main()函数是程序的第一个函数。从而有必要在程序的源代码中找到main()函数的位置，因此可以在那个点开始执行。(全局变量可能在main()函数的前面，因此，即使main()是第一个函数，也不一定是第一行代码)。由于预扫描程序找到了每个函数的入口点，它也就找到了main()函数的入口点。

执行预扫描的函数是prescan()。如下所示：

```
// Find the location of all functions in the program
// and store global variables.
```

```
void prescan()
{
  char *p, *tp;
  char temp[MAX_ID_LEN+1];
  token_ireps datatype;
  func_type ft;

  // When brace is 0, the current source position
  // is outside of any function.
  int brace = 0;

  p = prog;

  do {
    // Bypass code inside functions
    while(brace) {
      get_token();
      if(tok == END) throw InterpExc(UNBAL_BRACES);
      if(*token == '{') brace++;
      if(*token == '}') brace--;
    }

    tp = prog; // save current position
    get_token();

    // See if global var type or function return type.
    if(tok==CHAR || tok==INT) {
      datatype = tok; // save data type
      get_token();

      if(token_type == IDENTIFIER) {
        strcpy(temp, token);
        get_token();

        if(*token != '(') { // must be global var
          prog = tp; // return to start of declaration
          decl_global();
        }
        else if(*token == '(') { // must be a function

          // See if function already defined.
          for(unsigned i=0; i < func_table.size(); i++)
            if(!strcmp(func_table[i].func_name, temp))
              throw InterpExc(DUP_FUNC);

          ft.loc = prog;
          ft.ret_type = datatype;
          strcpy(ft.func_name, temp);
          func_table.push_back(ft);

          do {
            get_token();
```

```
        } while(*token != ')');
        // Next token will now be opening curly
        // brace of function.
       }
       else putback();
     }
    }
    else {
     if(*token == '{') brace++;
     if(*token == '}') brace--;
    }
  } while(tok != END);
  if(brace) throw InterpExc(UNBAL_BRACES);
  prog = p;
}
```

prescan()函数按如下的方式运行。每次遇到左花括号时，brace 增 1。每次碰到右花括号时，brace 减 1。因此，当 brace 大于 0 时，从函数中读取当前的令牌。然而，如果找到变量时，则 brace 等于 0，预扫描程序就知道这一定是一个全局变量。以相同的方式，如果 brace 等于 0 时遇到函数名，那么这一定是函数的定义(记住，Mini C++不支持函数原型)。

将全局变量存储在名为 global_vars 的向量中，这个向量具有如下所示的 var_type 类型的结构：

```
// This structure encapsulates the info
// associated with variables.
struct var_type {
  char var_name[MAX_ID_LEN+1]; // name
  token_ireps v_type; // data type
  int value; // value
};
```

这个结构存储了变量的名称、值以及类型。

全局变量由如下所示的 decl_global()函数存储到 global_vars 向量中：

```
// Declare a global variable.
void decl_global()
{
  token_ireps vartype;
  var_type vt;

  get_token(); // get type

  vartype = tok; // save var type

  // Process comma-separated list.
  do {
    vt.v_type = vartype;
    vt.value = 0; // init to 0
    get_token(); // get name

    // See if variable is a duplicate.
```

```
    for(unsigned i=0; i < global_vars.size(); i++)
      if(!strcmp(global_vars[i].var_name, token))
        throw InterpExc(DUP_VAR);

    strcpy(vt.var_name, token);
    global_vars.push_back(vt);

    get_token();
  } while(*token == ',');

  if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}
```

本质上，decl_global()函数获取变量的类型和名称，将其初始化为 0，然后放到 global_vars 的结尾。然而，首先要执行一个检查，以确保没有已经声明具有相同名称的变量。

每个用户自定义函数的位置被放入名为 func_table 的向量中，这个向量存储 func_type 类型的结构，如下所示：

```
// This structure encapsulates function info.
struct func_type {
  char func_name[MAX_ID_LEN+1]; // name
  token_ireps ret_type; // return type
  char *loc; // location of entry point in program
};
```

每个函数的入口都包含了返回类型、名称以及函数入口点在源代码中的位置。在函数存储到这个表格之前，prescan()函数检查是否存在相同名称的函数。注意没有存储参数信息。这个信息在运行时实际调用函数时获取。

9.6.3 interp()函数

interp()函数是解释程序的核心。正是这个函数基于输入流中的下一个令牌决定了采取的操作。这个函数用来解释一个代码单元并返回。如果这个单元由一条语句组成，那么这条语句被解释，interp()函数返回。然而，如果读取了一个左花括号，那么这个代码块中的所有语句都会被解释。这是由 block 标志管理的，如果读取了左花括号，则将 block 设置为 1。interp()函数一直解释后续的语句，直到读取右花括号。interp()函数如下所示：

```
// Interpret a single statement or block of code. When
// interp() returns from its initial call, the final
// brace (or a return) in main() has been encountered.
void interp()
{
  int value;
  int block = 0;

  do {
    // Don't interpret until break is handled.
    if(breakfound) return;

    token_type = get_token();
```

```
// See what kind of token is up.
if(token_type == IDENTIFIER ||
   *token == INC || *token == DEC)
{
  // Not a keyword, so process expression.
  putback(); // restore token to input stream for
             // further processing by eval_exp()
  eval_exp(value); // process the expression
  if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}
else if(token_type==BLOCK) { // block delimiter?
  if(*token == '{') { // is a block
    block = 1; // interpreting block, not statement
    // Record nested scope.
    nest_scope_stack.push(local_var_stack.size());
  }
  else { // is a }, so reset scope and return
    // Reset nested scope.
    local_var_stack.resize(nest_scope_stack.top());
    nest_scope_stack.pop();
    return;
  }
}
else // is keyword
  switch(tok) {
    case CHAR:
    case INT:     // declare local variables
      putback();
      decl_local();
      break;
    case RETURN: // return from function call
      func_ret();
      return;
    case IF: // process an if statement
      exec_if();
      break;
    case ELSE: // process an else statement
      find_eob(); // find end of else block
                  // and continue execution
      break;
    case WHILE: // process a while loop
      exec_while();
      break;
    case DO: // process a do-while loop
      exec_do();
      break;
    case FOR: // process a for loop
      exec_for();
      break;
    case BREAK: // handle break
      breakfound = true;
```

```
        // Reset nested scope.
        local_var_stack.resize(nest_scope_stack.top());
        nest_scope_stack.pop();
        return;
      case SWITCH: // handle a switch statement
        exec_switch();
        break;
      case COUT: // handle console output
        exec_cout();
        break;
      case CIN: // handle console input
        exec_cin();
        break;
      case END:
        exit(0);
    }
  } while (tok != END && block);
  return;
}
```

除了调用类似于exit()的函数之外，C++程序在遇到main()中的最后一个花括号(或者return)时结束，——而不一定在源代码的最后一行结束。这也是interp()只执行一条语句或者一个代码块而不是整个程序的原因之一。另外从概念上讲，C++由代码块组成。因此，每次遇到新代码块时，都会调用interp()函数。这包括函数块以及由各种C++语句开始的代码块，如if。在执行程序的过程中，Mini C++有可能递归调用interp()函数。

interp()函数的运行方式如下。首先，检查程序中是否有 break 语句。如果有一条，则全局变量breakfound为true。这个变量一直保持到被解释程序的其他部分清除，当描述控制语句的解释时，您将会看到这一点。

假定breakfound为false，则interp()函数从程序中读取下一个令牌。如果这个令牌是标识符，则这条语句一定是一个表达式，从而会调用表达式解析器。由于表达式解析器希望读取表达式本身的第一个令牌，通过调用putback()函数，这个令牌返回到输入流。当eval_exp()返回时，token将保存表达式解析器读取的最后一个令牌。如果token没有包含分号，就会报告错误。

如果从程序中读取的下一个令牌是左花括号，那么block设置为1，local_var_stack当前的大小被压入nest_scope_stack。local_var_stack用于保存所有的局部变量，包括在嵌套的块作用域中声明的变量。相反，如果下一个令牌是右花括号，则local_var_stack的大小缩短为nest_scope_stack顶部所指定的大小。从而有效地删除了这个作用域内声明的任何局部变量。因此，“{”会导致保存局部变量堆栈的大小，“}”导致局部变量堆栈重新设置为其原始大小。这个机制支持嵌套的局部作用域。

最后，如果这个令牌是一个关键字，则会执行switch语句，并调用合适的例程来处理这条语句。get_token()给定关键字的等价整型值的原因是为了能够使用switch语句，而不是使用涉及到字符串比较的一系列if语句(这样做相当慢)。

9.6.4 处理局部变量

当解释程序遇到int或者char关键字时，会调用decl_local()函数为局部变量创建存储空间。

前面讲过，一旦程序开始执行，解释程序就不会遇到全局变量的声明语句，因为只执行函数中的代码。因此，如果发现了变量声明语句，那么一定是一个局部变量的声明(或者参数，在下一部分讨论)。通常，局部变量存储在堆栈中。如果对语言进行编译，则通常使用系统堆栈。然而，在解释模式中，解释程序必须维护局部变量的堆栈。

在 Mini C++中，local_var_stack 为局部变量提供了一个堆栈。有些出人意料的是，local_var_stack 是一个 vector 容器而不是 stack 容器。原因是：尽管以堆栈的风格来管理局部变量，然而在需要访问变量值时，必须能够按顺序搜索容器。stack 容器不能方便地提供这种顺序搜索，而 vector 容器可以。

decl_local()函数如下所示：

```
// Declare a local variable.
void decl_local()
{
  var_type vt;

  get_token(); // get var type
  vt.v_type = tok; // store type

  vt.value = 0; // init var to 0

  // Process comma-separated list.
  do {
    get_token(); // get var name

    // See if variable is already the name
    // of a local variable in this scope.
    if(!local_var_stack.empty())
    for(int i=local_var_stack.size()-1;
        i >= nest_scope_stack.top(); i--)
    {
      if(!strcmp(local_var_stack[i].var_name, token))
        throw InterpExc(DUP_VAR);
    }

    strcpy(vt.var_name, token);
    local_var_stack.push_back(vt);
    get_token();
  } while(*token == ',');

  if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}
```

每次遇到局部变量时，都会将其名称、类型以及值(初始值为 0)压入堆栈。这个过程如下。decl_local()函数首先读取要声明的变量的类型，并将其初始值设置为 0。随后，进入一个循环，在循环内读取由逗号分割的标识符的名称。每循环一次，关于每个变量的信息都会压入到局部变量堆栈中。在这个过程中，进行了一个检查，以确保当前的作用域内不存在相同名称的变量。(我们发现，在当前作用域内声明的变量在 local_var_stack 的当前栈顶以及保存在 nest_scope_stack 栈顶的索引之间)。最后，检查最后的令牌，以确保它包含分号。

9.6.5 调用用户自定义的函数

为 C++实现一个解释程序最困难的部分可能是管理用户自定义函数的执行。解释程序不仅要在新的位置读取源代码，然后在函数结束后返回到调用例程，解释程序还必须处理 3 个任务：参数的传递、参数的分配以及函数的返回值。

所有的函数调用(除了最初的对 main()的调用)都是通过表达式解析器从 atom()函数中由 call()调用。实际上处理函数调用细节的是 call()函数。call()函数如下所示，让我们仔细分析一下：

```
// Call a function.
void call()
{
  char *loc, *temp;
  int lvartemp;

  // First, find entry point of function.
  loc = find_func(token);

  if(loc == NULL)
    throw InterpExc(FUNC_UNDEF); // function not defined
  else {
    // Save local var stack index.
    lvartemp = local_var_stack.size();

    get_args(); // get function arguments
    temp = prog; // save return location

    func_call_stack.push(lvartemp); // push local var index

    prog = loc; // reset prog to start of function
    get_params();  // load the function's parameters with
                   // the values of the arguments

    interp(); // interpret the function

    prog = temp; // reset the program pointer

    if(func_call_stack.empty()) throw InterpExc(RET_NOCALL);

    // Reset local_var_stack to its previous state.
    local_var_stack.resize(func_call_stack.top());
    func_call_stack.pop();
  }
}
```

call()函数做的第一件事情是通过调用 find_func()函数来寻找源代码中指定函数的入口点的位置。随后在 lvartemp 变量中保存了局部变量堆栈的当前大小。然后调用了 get_args()来处理任何函数参数。get_args()函数读取逗号分割的表达式列表，并且以相反的顺序将其压入局部变量堆栈中(以相反的顺序将表达式压入堆栈，当解释函数时，可以比较容易地与对应的参数匹配)。当这些值被压入时，没有给定名称。函数参数的名称是由 get_params()函数给定的，稍后将讨

论这个函数。

一旦完成函数参数的处理，prog 的当前值就被保存在 temp 中。这个位置是函数的返回点。随后，lvartemp 的值被压入函数调用堆栈 func_call_stack 中。其目的是在每次调用函数时，存储局部变量堆栈顶部的索引。这个值代表了与被调用函数相关的变量(以及参数)相关的局部变量堆栈的开始点。使用函数调用堆栈顶部的值阻止函数访问不是它声明的任何局部变量。

下面的两行代码将程序指针指向函数的开始，并调用 get_params()将函数形参的名称与保存在局部变量堆栈中的参数的值链接起来。为此，get_params()读取每个参数，并将其名称复制到 local_var_stack 中已经存储的对应的参数。

函数的实际执行是通过调用 interp()完成的。当 interp()返回时，程序指针(prog)重新设置为返回点，局部变量堆栈索引重新设置为函数调用前的值。最后一步有效地从堆栈中删除所有的函数局部变量和参数。

如果被调用的函数包含了 return 语句，那么 interp()会在返回到 call()函数之前调用 func_ret()。这个函数处理任何返回值，如下所示：

```
// Return from a function.
void func_ret()
{
  int value;

  value = 0;

  // Get return value, if any.
  eval_exp(value);

  ret_value = value;
}
```

全局变量 ret_value 是一个保存了函数返回值的整型数。一眼看上去，您可能会有些迷惑，为什么使用局部变量 value 而不是 ret_value 来接收 eval_exp()返回的值。原因是函数可能是递归的，eval_exp()可能需要调用相同的函数来获取它的值。在此情况下，全局变量有可能被改写，因此不能用来接收这个值。

9.6.6 给变量赋值

我们暂时返回到表达式解析器。当遇到赋值语句时，计算表达式右边的值，并调用 assign_var()将这个值赋给左边的变量。然而，C++支持各种作用域，包括全局作用域(正式的名称是命名空间作用域)和局部作用域。另外，局部作用域可以是嵌套的。这很重要，因为它影响到了 Mini C++寻找变量值的方式。为了理解这种方式，考虑下面的 Mini C++程序：

```
int count;

int main()
{
  int count, i;

  count = 100;
```

```
  i = f();

  return 0;
}

int f()
{
  int count;

  count = 99;

  return count;
}
```

当赋予变量 count 一个值时，assign_var()函数如何知道使用哪个变量呢？是全局的 count 变量还是局部的 count 变量？答案很简单。在 C++中，局部变量的优先级高于相同名称的全局变量的优先级。另外，局部变量在它的代码块之外不会被发现。为了观察如何使用这些规则来解决前面的赋值，检查如下所示的 assign_var()函数：

```
// Assign a value to a variable.
void assign_var(char *vname, int value)
{
  // First, see if it's a local variable.
  if(!local_var_stack.empty())
    for(int i=local_var_stack.size()-1;
        i >= func_call_stack.top(); i--)
    {
      if(!strcmp(local_var_stack[i].var_name,
                 vname))
      {
        if(local_var_stack[i].v_type == CHAR)
          local_var_stack[i].value = (char) value;
        else if(local_var_stack[i].v_type == INT)
          local_var_stack[i].value = value;
        return;
      }
    }

  // Otherwise, try global vars.
  for(unsigned i=0; i < global_vars.size(); i++)
    if(!strcmp(global_vars[i].var_name, vname)) {
      if(global_vars[i].v_type == CHAR)
        global_vars[i].value = (char) value;
      else if(global_vars[i].v_type == INT)
        global_vars[i].value = value;
      return;
    }

  throw InterpExc(NOT_VAR); // variable not found
}
```

正如前面部分所解释的那样，每次调用函数时，局部变量堆栈(local_var_stack)顶部的当前索引被压入函数调用堆栈(func_call_stack)。这意味着函数定义的任何局部变量(或者参数)在这个点上被压入堆栈。因此，assign_var()函数首先搜索 local_var_stack，从堆栈的当前栈顶开始，当索引到达最近一次的函数调用时停止。这个机制确保了只检查这个函数的局部变量。(另外这种机制还有助于支持递归函数，因为每次调用函数时，栈顶的当前值都会被保存)。因此，在这个示例中，main()函数中的这一行

```
count=100;
```

导致 assign_var()寻找 main()中的局部变量 count。在 f()中，语句

```
count=99;
```

导致 assign_var()寻找 f()中声明的 count。

如果没有局部变量与某个变量名称匹配，就会搜索全局变量的名称列表。如果局部变量和全局变量的列表都没有包含这个变量，就会发生语法错误。

现在考虑嵌套循环中声明的变量的情况，如下所示:

```
int f(int n)
{
  int count;

  count = 99

  if(n > 0) {
    int count; // this count is local to the if

    count = 100; // refers to count in if block
    // ...
  }

  return count; // refers to outer count.
}
```

在此情况下，外部变量 count(函数代码开始时声明的)首先被压入堆栈 local_var_stack。然后 if 代码块中的局部变量 count 被压入 local_var_stack。因此，当执行下面的语句时:

```
count=100;// refers to count in if block
```

assign_var()查找 count，并首先发现 if 代码块中的局部变量 count 的副本，正如它应当做的那样。

最后一点，在 interp()函数中，每当离开一个代码块时，local_var_stack 都会缩短到进入这个代码块之前的长度，从而有效地从堆栈中删除这个代码块中声明的所有局部变量。每次函数返回时，local_var_stack 也会缩短，从而删除了与函数相关的所有局部变量和参数。

9.6.7 执行 if 语句

既然已经讨论了 Mini C++解释程序的基本结构，现在是观察控制语句如何实现的时候了。通常，每次遇到关键字语句时，interp()就会调用处理这条语句的函数。解释各种控制语句的函

数都以前缀 exec_开头。例如，exec_ for ()函数解释 for，exec_switch()函数解释 switch 等。

最容易解释的控制语句之一是 if 语句。exec_if()函数处理 if 语句，代码如下所示：

```
// Execute an if statement.
void exec_if()
{
  int cond;

  eval_exp(cond); // get if expression.

  if(cond) { // if true, process target of IF
    // Confirm start of block.
    if(*token != '{')
      throw InterpExc(BRACE_EXPECTED);

    interp();
  }
  else {
    // Otherwise skip around IF block and
    // process the ELSE, if present.

    find_eob(); // find start of next line
    get_token();

    if(tok != ELSE) {
      // Restore token if no ELSE is present.
      putback();
      return;
    }

    // Confirm start of block.
    get_token();
    if(*token != '{')
      throw InterpExc(BRACE_EXPECTED);
    putback();

    interp();
  }
}
```

让我们仔细分析这个函数的操作。

首先，exec_if()调用 val_exp()求条件表达式的值。如果条件为 true(非 0)，则这个函数调用 interp()，interp()执行 if 代码块中的代码。如果条件为 false，则调用 find_eob()，这个函数立即将程序指针提前到 if 代码块结尾后的位置。如果存在 else 语句，则执行与 else 相关的代码块。否则，简单地开始下一行代码的执行。

如果执行 if 代码块，并且存在 else 代码块，则必须有某种方式来忽略掉 else 代码块。当遇到 else 语句时，interp()完成了这个任务。在此情况下，interp()简单地调用 find_eob()来忽略掉 else 代码块。在此之后，继续执行 else 代码块之后的第一条语句。记住，(在语法正确的程序中)只有在执行 if 代码块之后，interp()才会处理 else。

另外一点是：注意 exec_if()将确认 if 和 else 的目标代码包含在一个代码块中。正如前面所述，为了简化解释程序，所有控制语句的目标必须是代码块。该限制使解释程序代码块连接成一个整体。

9.6.8 switch 语句和 break 语句

解释 switch 语句需要的工作多于解释 if 语句。原因之一是它的语法更加复杂。另一个原因是它与 break 语句有关。因此，为了处理 switch 语句，意味着必须处理 break 语句。在此对二者进行了分析。

处理 break 语句相当容易，因为无论在 switch 语句还是在循环中使用，它都执行相同的操作：退出与这条语句相关的代码块。Mini C++用名为 breakfound 的全局标记变量来处理 break 语句，其初始值为 false。当发现 break 语句时，breakfound 设置为 true。然后用 switch 语句(以及后面提到的循环语句)来检查这个变量，以判断是否已经执行 break 语句。如果 breakfound 为 true，则终止当前的代码块，并将 breakfound 重新设置为 false。

break 语句由如下所示的 interp()中的代码处理：

```
case BREAK: // handle break
   breakfound = true;

  // Reset nested scope.
  local_var_stack.resize(nest_scope_stack.top());
  nest_scope_stack.pop();
  return;
```

正如您所看到的那样，除了将 breakfound 设置为 true 之外，局部变量堆栈被重新设置为被终止的代码块开始之前的状态。变量可以在任何代码块内声明，并局限于这个代码块。因此，当发现 break 时，与封闭的代码块相关的任何局部变量都必须被删除。

处理 switch 语句的 exec_switch()函数如下所示：

```
// Execute a switch statement.
void exec_switch()
{
  int sval, cval;
  int brace;

  eval_exp(sval); // Get switch expression.

  // Check for start of block.
  if(*token != '{')
    throw InterpExc(BRACE_EXPECTED);

  // Record new scope.
  nest_scope_stack.push(local_var_stack.size());

  // Now, check case statements.
  for(;;) {
    brace = 1;
    // Find a case statement.
```

```
    do {
      get_token();
      if(*token == '{') brace++;
      else if(*token == '}') brace--;
    } while(tok != CASE && tok != END && brace);

    // If no matching case found, then skip.
    if(!brace) break;

    if(tok == END) throw InterpExc(SYNTAX);

    // Get value of the case statement.
    eval_exp(cval);

    // Read and discard the :
    get_token();

    if(*token != ':')
      throw InterpExc(COLON_EXPECTED);

    // If values match, then interpret.
    if(cval == sval) {
      brace = 1;
      do {
        interp();

        if(*token == '{') brace++;
        else if(*token == '}') brace--;
      } while(!breakfound && tok != END && brace);

      // Find end of switch statement.
      while(brace) {
        get_token();
        if(*token == '{') brace++;
        else if(*token == '}') brace--;
      }
      breakfound = false;
      break;
    }
  }
}
```

首先，exec_switch()函数获取 switch 表达式的值，并将这个值存储在 sval 中。随后，它检查 switch 代码块的开始，并将局部变量堆栈的栈顶存储在 nest_scope_stack 中。这个步骤是必须的，因为 switch 语句创建了嵌套的作用域。随后，检查每个 case 语句的值，直到遇到与 sval 中的值匹配的 case 或者到达 switch 语句的结尾。(为了简单起见，Mini C++没有支持 default 语句)。如果找到了匹配 case，则执行与这个 case 相关的语句，直到遇到 break 语句(也就是说，直到 breakfound 为 true)，或者直到发现了 switch 代码块的结尾。在遇到 break 语句之后，找到 switch 代码块的结尾，则 exec_switch()函数结束，然后将 breakfound 设置为 false。

9.6.9 处理 while 循环

解释 while 循环相当容易。执行这个任务的函数是 exec_while()，代码如下所示：

```
// Execute a while loop.
void exec_while()
{
  int cond;
  char *temp;

  putback(); // put back the while
  temp = prog; // save location of top of while loop

  get_token();
  eval_exp(cond); // check the conditional expression

  // Confirm start of block.
  if(*token != '{')
    throw InterpExc(BRACE_EXPECTED);

  if(cond)
    interp(); // if true, interpret
  else { // otherwise, skip to end of loop
    find_eob();
    return;
  }

  prog = temp; // loop back to top

  // Check for break in loop.
  if(breakfound) {
    // Find start of loop block.
    do {
      get_token();
    } while(*token != '{' && tok != END);

    putback();
    breakfound = false;
    find_eob();
    return;
  }
}
```

exec_while()函数运行方式如下。首先，while 令牌被放回到输入流，while 在程序中的位置被保存在 temp 指针中。这个地址用来允许解释程序循环回 while 的顶部。随后，为了从输入流中删除 while，它被重新读取，并调用 eval_exp()来求 while 的条件表达式的值。如果条件表达式为 true，则调用 interp()来解释 while 代码块。当 interp()返回时，prog(程序指针)被设置为 while 循环开始的位置，这样做导致当控制返回到 interp()时，程序执行在循环的顶部重新启动，从而导致了循环的下一次迭代。然而，如果 interp()是因为在循环内发现了 break 语句而返回，迭代会终止，找到 while 代码块的结尾，exec_while()函数返回。当条件表达式为 false 时，while 代

码块的结尾被发现，函数返回。

9.6.10 处理 do-while 循环

do-while 循环的处理方式非常类似于 while。当 interp()遇到 do 语句的时候，调用 exec_do() 函数，代码如下所示：

```
// Execute a do loop.
void exec_do()
{
  int cond;
  char *temp;


  // Save location of top of do loop.
  putback(); // put back do
  temp = prog;

  get_token(); // get start of loop block

  // Confirm start of block.
  get_token();
  if(*token != '{')
    throw InterpExc(BRACE_EXPECTED);
  putback();

  interp(); // interpret loop

  // Check for break in loop.
  if(breakfound) {
    prog = temp;
    // Find start of loop block.
    do {
      get_token();
    } while(*token != '{' && tok != END);

    // Find end of while block.
    putback();
    find_eob();

    // Now, find end of while expression.
    do {
      get_token();
    } while(*token != ';' && tok != END);
    if(tok == END) throw InterpExc(SYNTAX);

    breakfound = false;
    return;
  }

  get_token();
  if(tok != WHILE) throw InterpExc(WHILE_EXPECTED);
```

```
  eval_exp(cond); // check the loop condition

  // If true loop; otherwise, continue on.
  if(cond) prog = temp;
}
```

do-while 循环和 while 循环之间的主要区别是，do-while 至少会执行一次它的代码块，因为条件表达式在循环的底部。因此，exec_do()首先将循环顶部的位置保存在 temp 中，然后调用 interp()来解释与这个循环相关的代码块。当 interp()返回时，获取对应的 while，并且求条件表达式的值。如果条件表达式为 true，prog 被重新设置为循环的顶部；否则，继续向下执行。如果遇到 break 语句，迭代终止，找到循环代码块的结尾。

9.6.11 for 循环

相对于其他循环而言，对 for 循环的解释是一个更加艰难的挑战。部分原因是 C++语言 for 循环的结构设计考虑了编译。主要问题是必须在每个循环的顶部检查 for 循环的条件表达式，而增量部分发生在循环的底部。因此，虽然 for 循环的这两个部分在源代码中连续出现，但它们的解释却被迭代的代码块分开。然而，多花一点力气，for 循环还是可以被正确地解释的。

当 interp()遇到 for 语句时，会调用 exec_for()函数。函数代码如下所示：

```
// Execute a for loop.
void exec_for()
{
  int cond;
  char *temp, *temp2;
  int paren ;

  get_token(); // skip opening (
  eval_exp(cond); // initialization expression

  if(*token != ';') throw InterpExc(SEMI_EXPECTED);
  prog++; // get past the ;
  temp = prog;

  for(;;) {
    // Get the value of the conditional expression.
    eval_exp(cond);

    if(*token != ';') throw InterpExc(SEMI_EXPECTED);
    prog++; // get past the ;
    temp2 = prog;

    // Find start of for block.
    paren = 1;
    while(paren) {
      get_token();
      if(*token == '(') paren++;
      if(*token == ')') paren--;
    }
```

```
    // Confirm start of block.
    get_token();
    if(*token != '{')
      throw InterpExc(BRACE_EXPECTED);
    putback();

    // If condition is true, interpret
    if(cond)
      interp();
    else { // otherwise, skip to end of loop
      find_eob();
      return;
    }

    prog = temp2; // go to increment expression

    // Check for break in loop.
    if(breakfound) {
      // Find start of loop block.
      do {
        get_token();
      } while(*token != '{' && tok != END);

      putback();
      breakfound = false;
      find_eob();
      return;
    }

    // Evaluate the increment expression.
    eval_exp(cond);

    prog = temp; // loop back to top
  }
}
```

这个函数开始处理 for 中的初始化表达式。for 的初始化部分只执行一次，并不构成循环的一部分。随后，程序指针被提前到初始化表达式结束的分号之后，这个地址赋给了 temp。这个位置是条件表达式的开始。然后，进入一个无限的循环，检查循环的条件部分，并将增量表达式的起始地址赋给 temp2。然后，找到循环开始的代码。最后，如果条件表达式为 true，则解释循环代码块。否则，找到代码块的结尾，继续执行在 for 循环之后的语句。假定循环执行，当对 interp()的调用返回时，增加表达式被求值，这个过程重复。当然，如果在循环代码块中遇到了 break 语句，这个过程结束。

9.6.12 处理 cin 和 cout 语句

由于通过 cin 和 cout 的 I/O 是 C++的基础部分，因此 Mini C++必须支持它们。然而，*Mini C++* 并没有用商业化编译器处理 cin 和 cout 的方式来处理 I/O 链接。您知道，cin 和 cout 是预定义的标识符，这两个标识符对应于与标准输入和标准输出相链接的流。通过使用 I/O 运算符“<<”

和“>>”，它们可以在控制台输入和输出信息。因此，“<<”以及“>>”为了I/O而重载。然而，Mini C++不支持运算符重载。事实上，为了使得Mini C++尽可能的简单，甚至没有支持“<<”和“>>”移位运算符(然而，get_token()能够识别这些运算符)。尽管有这些限制，但是解释cin和cout语句相当容易。

通过cout的控制台输出由exec_cout()函数处理，代码如下所示：

```
// Execute a cout statement.
void exec_cout()
{
  int val;

  get_token();
  if(*token != LS) throw InterpExc(SYNTAX);

  do {
    get_token();

    if(token_type==STRING) {
      // Output a string.
      cout << token;
    }
    else if(token_type == NUMBER ||
            token_type == IDENTIFIER) {
      // Output a number.
      putback();
      eval_exp(val);
      cout << val;
    }
    else if(*token == '\'') {
      // Output a character constant.
      putback();
      eval_exp(val);
      cout << (char) val;
    }

    get_token();
  } while(*token == LS);

  if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}
```

当遇到cout标识符时，会读取下一个令牌。如果下一个令牌不是“<<”，就会报告语法错误。否则，就会进入一个循环，然后输出“<<”右边的字符串或者表达式的值。这个过程持续到cout语句的结尾。

cin语句由exec_cin()函数处理，代码如下所示：

```
// Execute a cin statement.
void exec_cin()
{
  int val;
```

```
  char chval;
  token_ireps vtype;

  get_token();
  if(*token != RS) throw InterpExc(SYNTAX);

  do {
    get_token();
    if(token_type != IDENTIFIER)
      throw InterpExc(NOT_VAR);

      vtype = find_var_type(token);

      if(vtype == CHAR) {
        cin >> chval;
        assign_var(token, chval);
      }
      else if(vtype == INT) {
        cin >> val;
        assign_var(token, val);
      }

      get_token();
  } while(*token == RS);

  if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}
```

当遇到 cin 标识符时，会读取下一个令牌。如果下一个令牌不是“>>”，则会报告语法错误。否则，就会进入一个循环，获取接收输入的变量，从控制台读取输入，并将这个输入存储在变量中。注意，判断整型或者字符型数据的变量类型被读取。这个过程持续到 cin 语句的结尾。

9.7 Mini C++的库函数

由于 Mini C++执行的程序不会被编译并链接，因此它使用的任何库例程都必须被 Mini C++处理。为此，最好的方法是创建一个接口函数，当遇到库函数时 Mini C++就调用这个接口函数。接口函数创建对实际的库函数的调用，并处理所有的返回值。

由于空间的限制，Mini C++只支持 4 个“库”函数：getchar()、putchar()、abs()以及 rand()。这些函数被转换为同名库函数的调用。Mini C++库例程在文件 libcpp.cpp 中。如下所示：

```
// ***** Internal Library Functions *****

// Add more of your own, here.

#include <iostream>
#include <cstdlib>
#include <cstdio>
#include "mccommon.h"
```

```
using namespace std;

// Read a character from the console.
// If your compiler supplies an unbuffered
// character intput function, feel free to
// substitute it for the call to cin.get().
int call_getchar()
{
  char ch;

  ch = getchar();

  // Advance past ()
  get_token();
  if(*token != '(')
    throw InterpExc(PAREN_EXPECTED);

  get_token();
  if(*token != ')')
    throw InterpExc(PAREN_EXPECTED);

  return ch;
}

// Write a character to the display.
int call_putchar()
{
  int value;

  eval_exp(value);

  putchar(value);

  return value;
}

// Return absolute value.
int call_abs()
{
  int val;

  eval_exp(val);

  val = abs(val);

  return val;
}

// Return a random integer.
int call_rand()
{
```

```
  // Advance past ()
  get_token();
  if(*token != '(')
    throw InterpExc(PAREN_EXPECTED);

  get_token();
  if(*token != ')')
    throw InterpExc(PAREN_EXPECTED);

  return rand();
}
```

为了添加您选择的更多的库函数，首先将它们的名称和它们接口函数的地址键入到intern_func 数组中(在 parser.cpp 中声明)。随后，按照前面所示的函数，创建合适的接口函数。最后，在 mccommon.h 中加入它们的原型。

9.8 mccommon.h 头文件

Mini C++的 3 个源文件，minicpp.cpp, parser.cpp 以及 libcpp.cpp，包括头文件 mccommon.h，如下所示：

```
// Common declarations used by parser.cpp, minicpp.cpp,
// or libcpp.cpp, or by other files that you might add.
//
const int MAX_T_LEN = 128; // max token length
const int MAX_ID_LEN = 31; // max identifier length
const int PROG_SIZE = 10000; // max program size
const int NUM_PARAMS = 31; // max number of parameters

// Enumeration of token types.
enum tok_types { UNDEFTT, DELIMITER, IDENTIFIER,
                 NUMBER, KEYWORD, TEMP, STRING, BLOCK };

// Enumeration of internal representation of tokens.
enum token_ireps { UNDEFTOK, ARG, CHAR, INT, SWITCH,
                   CASE, IF, ELSE, FOR, DO, WHILE, BREAK,
                   RETURN, COUT, CIN, END };

// Enumeration of two-character operators, such as <=.
enum double_ops { LT=1, LE, GT, GE, EQ, NE, LS, RS, INC, DEC };

// These are the constants used when throwing a
// syntax error exception.
//
// NOTE: SYNTAX is a generic error message used when
// nothing else seems appropriate.
enum error_msg
     { SYNTAX, NO_EXP, NOT_VAR, DUP_VAR, DUP_FUNC,
       SEMI_EXPECTED, UNBAL_BRACES, FUNC_UNDEF,
```

```
        TYPE_EXPECTED, RET_NOCALL, PAREN_EXPECTED,
        WHILE_EXPECTED, QUOTE_EXPECTED, DIV_BY_ZERO,
        BRACE_EXPECTED, COLON_EXPECTED };

extern char *prog; // current location in source code
extern char *p_buf; // points to start of program buffer

extern char token[MAX_T_LEN+1]; // string version of token
extern tok_types token_type; // contains type of token
extern token_ireps tok; // internal representation of token

extern int ret_value; // function return value
extern bool breakfound; // true if break encountered

// Exception class for Mini C++.
class InterpExc {
  error_msg err;
public:
  InterpExc(error_msg e) { err = e; }
  error_msg get_err() { return err; }
};

// Interpreter prototypes.
void prescan();
void decl_global();
void call();
void putback();
void decl_local();
void exec_if();
void find_eob();
void exec_for();
void exec_switch();
void get_params();
void get_args();
void exec_while();
void exec_do();
void exec_cout();
void exec_cin();
void assign_var(char *var_name, int value);
bool load_program(char *p, char *fname);
int find_var(char *s);
void interp();
void func_ret();
char *find_func(char *name);
bool is_var(char *s);
token_ireps find_var_type(char *s);

// Parser prototypes.
void eval_exp(int &value);
void eval_exp0(int &value);
void eval_exp1(int &value);
void eval_exp2(int &value);
```

```
void eval_exp3(int &value);
void eval_exp4(int &value);
void eval_exp5(int &value);
void atom(int &value);
void sntx_err(error_msg error);
void putback();
bool isdelim(char c);
token_ireps look_up(char *s);
int find_var(char *s);
tok_types get_token();
int internal_func(char *s);
bool is_var(char *s);

// "Standard library" prototypes.
int call_getchar();
int call_putchar();
int call_abs();
int call_rand();
```

9.9 编译并链接 Mini C++解释程序

为了使用 Mini C++，必须编译并链接 minicpp.cpp, parser.cpp 以及 libcpp.cpp。您几乎可以使用任何流行的 C++编译器，包括 Borland C++和 Visual C++。例如，对于 Visual C++，您可以使用如下的命令行：

```
cl -GX minicpp.cpp parser.cpp libcpp.cpp
```

对于 Borland C++，可以使用如下的命令行：

```
bcc32 minicpp.cpp parser.cpp libcpp.cpp
```

如果您使用了不同的编译器，只需要简单地遵循它附带的指示。

提示：

对于 Visual C++的比较老的版本，可能没有给予 Mini C++足够的堆栈空间。您可以使用 /Fsize 选项来增加堆栈。

为了运行一个程序，需要在命令行的 minicpp 之后指定它的名称。例如，通过下面的命令行运行名为 test.cpp 的程序：

```
minicpp test.cpp
```

9.10 演示 Mini C++

本节给出了一些 C++程序来演示 Mini C++的特性和功能。第一个程序演示了 Mini C++支持的所有特性：

```
/* Mini C++ Demonstration Program #1.
```

```
   This program demonstrates all features
   of C++ that are recognized by Mini C++.
*/

int i, j; // global vars
char ch;

int main()
{
  int i, j; // local vars

  // Call a "standard library' function.
  cout << "Mini C++ Demo Program.\n\n";

  // Call a programmer-defined function.
  print_alpha();

  cout << "\n";

  // Demonstrate do and for loops.
  cout << "Use loops.\n";
  do {
    cout << "Enter a number (0 to quit): ";
    cin >> i;

    // Demonstrate the if
    if(i < 0 ) {
      cout << "Numbers must be positive, try again.\n";
    }
    else {
      for(j = 0; j <= i; ++j) {
        cout << j << " summed is ";
       .cout << sum(j) << "\n";
      }
    }
  } while(i != 0);

  cout << "\n";

  // Demonstrate the break in a loop.
  cout << "Break from a loop.\n";
  for(i=0; i < 100; i++) {
    cout << i << "\n";
    if(i == 5) {
      cout << "Breaking out of loop.\n";
      break;
    }
  }

  cout << "\n";
```

```
  // Demonstrate the switch
  cout << "Use a switch.\n";
  for(i=0; i < 6; i++) {
    switch(i) {
      case 1: // can stack cases
      case 0:
        cout << "1 or 0\n";
        break;
      case 2:
        cout << "two\n";
        break;
      case 3:
        cout << "three\n";
        break;
      case 4:
        cout << "four\n";
        cout << "4 * 4 is "<< 4*4 << "\n";
//        break; // this break is optional
        // no case for 5
    }
  }
  cout << "\n";

  cout << "Use a library function to generate "
       << "10 random integers.\n";
  for(i=0; i < 10; i++) {
    cout << rand() << " ";
  }

  cout << "\n";
  cout << "Done!\n";

  return 0;
}

// Sum the values between 0 and num.
// This function takes a parameter.
int sum(int num)
{
  int running_sum;
  running_sum = 0;

  while(num) {
    running_sum = running_sum + num;
    num--;
  }
  return running_sum;
}

// Print the alphabet.
int print_alpha()
{
```

```
  cout << "This is the alphabet:\n";

  for(ch = 'A'; ch<='Z'; ch++) {
    putchar(ch);
  }
  cout << "\n";

  return 0;
}
```

下面是运行的样本：

```
Mini C++ Demo Program.

This is the alphabet:
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Use loops.
Enter a number (0 to quit): 10
0 summed is 0
1 summed is 1
2 summed is 3
3 summed is 6
4 summed is 10
5 summed is 15
6 summed is 21
7 summed is 28
8 summed is 36
9 summed is 45
10 summed is 55
Enter a number (0 to quit): 0
0 summed is 0

Break from a loop.
0
1
2
3
4
5
Breaking out of loop.

Use a switch.
1 or 0
1 or 0
two
three
four
4 * 4 is 16

Use a library function to generate 10 random integers.
130 10982 1090 11656 7117 17595 6415 22948 31126 9004
```

```
Done!
```

下面的程序演示了嵌套循环：

```
// Nested loop example.
int main()
{
  int i, j, k;

  for(i = 0; i < 5; i = i + 1) {
    for(j = 0; j < 3; j = j + 1) {
      for(k = 3; k ; k = k - 1) {
        cout << i <<", ";
        cout << j << ", ";
        cout << k << "\n";
      }
    }
  }

  cout << "done";

  return 0;
}
```

这个程序的部分输出如下所示：

```
0, 0, 3
0, 0, 2
0, 0, 1
0, 1, 3
0, 1, 2
0, 1, 1
0, 2, 3
0, 2, 2
0, 2, 1
.
.
.
```

下一个程序练习了赋值运算符：

```
// Assignments as operations.
int main()
{
  int a, b;

  a = b = 5;

  cout << a << " " << b << "\n";

  while(a=a-1) {
    cout << a << " ";
    do {
```

```
    cout << b << " ";
  } while((b=b-1) > -5);
  cout << "\n";
 }

 return 0;
}
```

这个程序的输出如下所示:

```
5 5
4 5 4 3 2 1 0 -1 -2 -3 -4
3 -5
2 -6
1 -7
```

下一个程序演示了递归函数。其中，函数 factr()用于计算某个数的阶乘。

```
// This program demonstrates a recursive function.

// A recursive function that returns the
// factorial of i.
int factr(int i)
{
 if(i<2) {
  return 1;
 }
 else {
    return i * factr(i-1);
 }
}

int main()
{
 cout << "Factorial of 4 is: ";
 cout << factr(4) << "\n";

 cout << "Factorial of 6 is: ";
 cout << factr(6) << "\n";

 return 0;
}
```

输出如下：

```
Factorial of 4 is: 24
Factorial of 6 is: 720
```

下面的程序完整地演示了函数的参数：

```
// A more rigorous example of function arguments.

int f1(int a, int b)
```

```
{
  int count;

  cout << "Args for f1 are ";
  cout << a << " " << b << "\n";

  count = a;
  do {
    cout << count << " ";
  } while(count=count-1);

  cout << a << " " << b
       << " " << a*b << "\n";

  return a*b;
}

int f2(int a, int x, int y)
{
  cout << "Args for f2 are ";
  cout << a << " " << x << " "
       << y << "\n";
  cout << x / a << " ";
  cout << y*x << "\n";

  return 0;
}

int main()
{
  f2(10, f1(10, 20), 99);

  return 0;
}
```

这个程序的输出如下所示：

```
Args for f1 are 10 20
10 9 8 7 6 5 4 3 2 1 10 20 200
Args for f2 are 10 200 99
20 19800
```

下面的程序练习了各种循环语句：

```
// Exercise the loop statements.
int main()
{
  int a;
  char ch;

  // The while.
  cout << "Enter a number: ";
  cin >> a;
```

```
  while(a) {
    cout << a*a << " ";
    --a;
  }
  cout << "\n";

  // The do-while.
  cout << "\nEnter characters, 'q' to quit.\n";
  do {
    // Use two "standard library" functions.
    ch = getchar();
    putchar(ch);
  } while(ch != 'q');
  cout << "\n\n";

  // the for.
  for(a=0; a<10; ++a) {
     cout << a << " ";
  }

  cout << "\n\nDone!\n";

  return 0;
}
```

这是一个运行的样本：

```
Enter a number: 10
100 81 64 49 36 25 16 9 4 1

Enter characters, 'q' to quit.
This is a test. q
This is a test. q

0 1 2 3 4 5 6 7 8 9

Done!
```

注意，在这个程序的运行中，内建的库函数 getchar()是行缓存的，从而只有按下 ENTER 时，putchar()才会显示字符。这种行为是 Mini C++调用库函数 getchar()的结果。您知道，大多数编译器实现的 getchar()都是行缓存的。关键是内建的、Mini C++函数展示了与底层的库函数相同的行为。

最后一个程序演示了嵌套作用域的使用。在这个程序中，变量 x 声明了 3 次：第 1 次是作为全局变量，第 2 次是作为 if 代码块的局部变量，第 3 次是在 while 代码块中声明。3 个变量是独立的，彼此各不相同。

```
// Demonstrate nested scopes.

int x; // global x

int main()
```

```
{
  int i;

  i = 4;

  x = 99; // global x is 99

  if(i == 4) {
    int x; // local x
    int num; // local to if statement

    x = i * 2;
    cout << "Outer local x before loop: "
         << x << "\n";

    while(x--) {
      int x; // another local x

      x = 18;
      cout << "Inner local x: " << x << "\n";
    }

    cout << "Outer local x after loop: "
         << x << "\n";
  }

  // Can't refer to num here because it is local
  // to the preceding if block.
// num = 10;

  cout << "Global x: " << x << "\n";
}
```

输出显示如下：

```
Outer local x before loop: 8
Inner local x: 18
Inner local x: 18
Inner local x: 18
Inner local x: 18
Inner local x: 18
Inner local x: 18
Inner local x: 18
Inner local x: 18
Outer local x after loop: -1
Global x: 99
```

9.11 改进 Mini C++

在设计 Mini C++时考虑了操作的透明性。目的是以最小的代价开发一个易于理解的解释程

序。这个解释程序还是以最容易扩展的方式设计的。因此，Mini C++不是特别快速或者高效。然而，这个解释程序的基本结构是正确的，可以使用如下步骤增加执行速度。

事实上所有的商业化解释程序都扩展了预扫描程序的任务。被解释的整个源程序从人类可以理解的方式转化为内部格式。在这种内部格式中，除了被引用的字符串和常量之外，所有的内容都转换为整型令牌，类似于 Mini C++将 C++的关键字转换为整型令牌的方式。Mini C++执行大量的字符串比较时，就可能这样做。例如，每次寻找变量或者函数时，都会发生多次的字符串比较。字符串的比较非常耗时，然而，如果源程序中的每个标识符都转换为整型数，就可以使用比较快速的整型数比较。通常，源程序到内部格式的转换是一个最重要的转换。可以将其应用于 Mini C++来提高它的效率。速度大为增加。

另一个改进的地方是为变量和函数查找例程，对于大程序尤其有意义。即使您将这些条目转换为整型令牌，当前搜索它们的方法也取决于有序的搜索。然而，可以用其他比较快速的方法来取代。例如，可以尝试使用 map 容器，或者可以使用某种散列方法或者树结构。

如前所述，相对于完整的 C++语法，Mini C++的一个限制是控制语句的实现，如 if，必须是用花括号包含的一个代码块。这样做的原因是它大大地简化了 find_eob()函数，这个函数在控制语句执行之后，寻找代码块的结尾。find_eob()函数只需要寻找与代码块开始的左花括号匹配的右花括号。您可能会发现，删除这个限制很有趣。

9.12 扩展 Mini C++

有两个一般的领域，可以扩展并增强 Mini C++解释程序：C++特性和辅助特性。下面的部分将就此进行简单讨论。

9.12.1 添加新的 C++特性

可以向 Mini C++添加两类基本的语句。第一类是附加转向语句，如 goto 和 continue 语句。您可能还想向 switch 中添加对 default 语句的支持。如果学习了 Mini C++解释其他语句的方式，添加这些特性几乎不会遇到麻烦。如果某些事物在第一时间没有运行，则可以通过显示处理过的每个令牌的内容来发现问题。

可以添加的第二类语句是对附加数据类型的支持。Mini C++已经为附加的数据类型包含了基本的“钩子”。例如，var_type 结构已经为这种类型的变量包含了一个字段。为了加入其他内建的类型(例如，float, double 以及 long)，只需要将这个值字段的大小提高到您想要保存的最大元素的长度。

添加类是一个更大的挑战。首先，必须定义实例化对象的方式。为此，必须分配足够大的内存来保存类的数据成员，并且将这块内存的引用存储到另一个新的字段，需要在 var_type 中添加这个字段。还需要处理 public 和 private 的概念。

对指针的支持并不比对其他数据类型的支持更加困难。然而，需要在表达式解析器中加入对指针运算符的支持。一旦实现了指针，数组就变得容易了。任何数组的空间都应该使用 new 来动态分配，为此，指向数组的指针应该被存储在一个新的字段中，这个字段应该添加到 var_type 中。

为了处理函数的不同返回类型，需要使用 func_type 结构中的 ret_type 字段。这个字段定义

了函数返回的数据的类型。该字段目前被设定，但是还未使用。

加入对#include 的支持是比较容易的。这个预处理程序指示可以很容易地在预扫描过程中处理。

最后一点：如果您想要试验语言的结构，不要害怕添加非 C++的扩展。例如，您可以很容易地添加第 4 章讨论的 foreach 循环。

9.12.2 添加辅助特性

解释程序给了您添加一些有趣并有用的特性的机会。例如，可以加入跟踪工具，来显示每个执行中的令牌。还可以加入在程序执行时，显示每个变量内容的功能。另一个您想加入的特性是一个完整的编辑器，从而可以“编辑并运行”，而不是使用其他编辑器来创建您的 C++程序。

了函数返回的数据的类型。该字段目前被设定，但是还未使用。

加入对#include 的支持是比较容易的。这个预处理程序指示可以很容易地在预扫描过程中处理。

最后一点：如果您想要试验语言的结构，不要害怕添加非 C++的扩展。例如，您可以很容易地添加第 4 章讨论的 foreach 循环。

9.12.2 添加辅助特性

解释程序给了您添加一些有趣并有用的特性的机会。例如，可以加入跟踪工具，来显示每个执行中的令牌。还可以加入在程序执行时，显示每个变量内容的功能。另一个您想加入的特性是一个完整的编辑器，从而可以“编辑并运行”，而不是使用其他编辑器来创建您的 C++程序。