计算机知识普及系列丛书

用C语言开发高级应用程序

周鸿仪　魏　东　编写
杨　帆　　审校

学苑出版社

用C语言开发高级应用程序

周鸿仪　魏　东　编写
杨　帆　　审校

学苑出版社
1993.

(京)新登字 151 号

内 容 提 要

本书每章内容的选择都是基于实用、有趣、独特或娱乐。

我们可以说，任何 C 程序员都能阅读本书。如果您刚刚入门，您可以照搬地使用这些程序直到您成为一名高手。如果您已是一个经验丰富的程序员，您可以把这些函数例程作为您自己应用的起点。总之，我们相信本书会给您带来新意和进步。

欲购本书的用户，请直接与北京 8721 信箱联系，电话 2562329，邮码 100080。

计算机知识普及系列丛书

用 C 语言开发高级应用程序

编　　写：周鸿仪　魏　东
审　　校：杨　帆
责任编辑：甄国宪
出版发行：学苑出版社　　**邮政编码**：100032
社　　址：北京市西城区成方街 33 号
印　　刷：兰空印刷厂
开　　本：787×1092　1/16
印　　张：23.625　　**字 数**：560 千字
印　　数：1～3000 册
版　　次：1993 年 12 月北京第 1 版第 1 次
ISBN7－5077－0821－7/TP・19
本册定价：17.00 元

目　　录

第一章　C语言解释器

开发语言解释器是一项有趣的工作，而对C语言程序员来说，又有什么能比开发一个C语言解释器更具有吸引力呢？

考虑要用一个所有C语言程序员都感兴趣的题目作为本书的开始。同时，这个题目又要是时兴的，并且要有实用价值。经过再三选择，最后终于决定开发一个小c语言解释器以作为本书的开端。

虽然编译器是非常重要又有价值的，但开发一个编译器是一个困难而漫长的过程。事实上，仅仅是编译器的运行库的建立就是一项繁重的任务。相反，开发一个语言解释器则是比较简单且容易管理的工作。同时，如果设计得当，解释器的运行也比编译器更容易理解。除了容易开发以外，语言解释器还具有编译器所没有的一个特性——能真正执行程序的机制。请记住：编译器仅仅把你的程序源代码翻译成计算机可以执行的形式，而解释器是真正在执行程序。正是这个差别使得解释器更具有吸引力。

如果你同大多数C语言程序员一样，那么你使用C语言就不仅仅是因为它具有较强的功能和灵活性，而且还由于对C语言本身就体现出一种美。事实上，由于C语言的简练和一致性，人们常把它称为“优美的”语言。许多有关C语言的文献资料都采取“从外向内看”的方式介绍C语言，很少从C语言“内部”揭示其特性。因此，有什能比用C语言解释执行C语言更适合于作为本书的开始呢？

在本章中，开发了一个能执行ANSI标准C的子语言的解释器。该解释器不仅能正常工作，而且经过了精心设计——你可以很容易地增强其功能，对它进行扩充，甚至可以加入ANSI C中所没有的特性。如果你以前未曾考虑过C是如何运行的，那么你将会惊喜地发现这是非常简单直观的。C语言是至今为止开发的计算机语言中最具有一致性的语言。完成本章后，你不仅将拥有一个可以使用和扩充其功能的C语言解释器，而且你将对C语言本身的内部结构获得进一步的了解。

注：本章中列出的C解释器的源代码非常长，但请别担心，如果你读完了本章中的论述部分，理解并执行这些程序就不会有太大问题。

1.1　解释器在实际应用中的重要性

虽然，小C语言解释器本身具有一定趣味性，语言解释器在实际计算中确有其一定的重要性。

同你可能知道的一样，C通常是一种“编译语言”，其主要原因是因为C是用来开发商业销售程序的语言，商业软件产品采用编译后的代码是由于它可以保护源代码，防止用户修改源代码，使程序能最有效地使用宿主计算机，以及其它原因。坦率地说，编译器将一直主宰商品软件的开发，也理应如此；但是，任何计算机语言既可编译也可解释。事实上，最近几年，有一些C语言解释器已出现在市场上了。

人们使用C解释器通常有两个原因。首先，C语言的初学者更愿意使用解释器的交互式环境，（虽然由于引入了C集成化编程环境，例如Borland的Turbo C和Microsoft的Quick C，这种情况正逐渐在改变）。

第二个原因是由于调试的需要。在调试中，解释器较之编译器的优点是在任何时候都可以知道并能修改所有变量的内容。而且，C解释器还提供了跟踪机制，这在编译环境中是比较困难的。

解释器的另一个实用性在于你可以把它作为一个数据库查询语言的基础，实际上由于其要执行任务的性质，所有的数据库查询语言都是解释执行的。虽然在大多数环境中用C作为查询语言不一定合适，但你学到的许多原理都是适用的。

语言解释器之所以吸引人还有另外一个原因：他们很容易修改，变换及补充。这意味着如果你想创建、试验、管理自己的语言，那么使用解释器比编译器更容易完成这些工作，解释器可以作为语言的原型环境，因为你可以改变语言的执行方式，并能很快看到结果。

1.2 Little C语言的规范说明

虽然ANSI C只有32个关键字（内部命令）。但C仍是一个具有很丰富的功能的语言。要想完整地描述和实现一个对整个C的解释器，远非在一个章节的篇幅内所能办到的。小C语言解释器能解释C语言的一个小子集。虽然如此，这个特定子集却包含了C语言的许多最重要的方面。该子集中所包含的内容主要根据以下两条准则来确定：

1.该特性是否是C语中最基本的，不可缺少的特性。

2.该特性是否是显示C语言的某一重要方面所必需的。

例如：递归函数、全局和局部变量这些特性都符合上述两准则。小C(今后用Little C)语言解释器支持三种循环结构（并不因为第一条准则，而是由于第二条）。然而，Little C不支持switch语句，因为它既不是必不可少的，也不能实现if语句（被支持的）所不能实现的功能。（switch的实现留待你自己练习用。）

由于这些原因，在Little C解释器中，我实现了C语言的以下特性：

- 具有局部变量、带参数的函数
- 递归
- if语句
- do-while，while和for循环
- 整数和字符变量
- 全局变量
- 整数和字符常量
- 字符串常量（有限的功能）
- 带及不带值的return语句
- 有限几个标准库函数
- 操作符：+，−，*，/，%，<，>，<=，>=，==，!=，单目−，和单目+，
- 返回整数的函数
- 注释

即使这个列表显得很短，也要用相当数量的代码来实现上述特性。这里的一个原因是由于要解释象C这样的结构化语言，必须先要付一定的“入场费”。

- Little C的一个重要约束条件

Little C解释器的源代码非常长，为了简化和缩短Little C的源代码，对C的语法加了一个小小的限制条件：if、while、do和for的目标码必须是由大括号{、}括起来的代码块，你不能只用一个简单语句，例如Little C无法正确解释下述代码：

```
for ( a=0; a<10; a=a+1 )
  for ( b=0; b<10; b=b+1 )
    for ( c=0; c<10; c=c+1 )
      puts ( "hi" );
if ( … )
  if ( … ) statement;
```

你应该如此写上述代码：

```
  for ( a=0; a<10; a=a+1 )  {
    for ( b=0; b<10; b=b+1 )  {
      for ( c=0; c<10; c=c+1 )  {
        puts ( "hi" );
      }
    }
  }
    if ( … ) {
      if ( … ) {
        statement;
      }
    }
```

这个约束使得解释器比较容易找到构成程序控制语句的目标代码的结尾之处。由于程序控制语句的目标码通常都是代码块，因此这个约束条件并不太苛刻（如果你愿意的话，稍加努力，就可以去掉这个约束条件）。

1.3 解释执行结构化语言

正如你所知道的，C是结构化的，它允许具有局部变量的独立的子程序，它也支持递归。你也可能会发现，在有些领域，为结构化语言写一个编译器要比写一个解释器更容易。例如，当一个编译器生成调用函数的代码时，它只需将调用参数压入系统堆栈中，然后对该函数执行机器指令CALL即可。返回时，函数将返回值放入CPU的累加器中，清除堆栈，然后执行机器指令RET。然而，当解释器“调用”一个函数时，它必须停下正在运行的程序，保存当前状态，找到待调用函数的位置，执行该函数，保存返回值，并且回到原来的位置，恢复旧的环境。（在后面的解释器中你会看到这一点的。）基本上，解释器必须模仿机器语言CALL和RETURN。并且，虽然在编译语言中递归比较容易处理，但在解释时，则需要花费一定的功夫。

1.4 关于C的非形式化说明

在我们开发C解释器之前，有必要了解C是如何构造的。如果你已经读过C语言的形式化规范说明（例如象ANSI标准规范），你一定知道这些说明相当冗长而且含义隐晦。请别着急—我们设计解释器时没必要如此形式化地去处理，因为C语言的大部分内容都是很直观的。虽然在商品化的编译器和解释器的开发中，语言的形式化规范是必不可少的，但开发Little C解释器却不必如此。（坦率地说，在本章中也没注解释如何理解C的形式化语法定义：这需要占用一本书的篇幅）。

虽然实现和理解本章中开发的Little C解释器并不要求你达到语言专家的水平，但你也必须对C是如何定义的有些基本知识。在我们这里，以下论述的知识就足够了。（你若想了解更形式化的论述，可以参阅ANSI的C语言标准，若想对语言有更完善的理论上的认识，可以参见《The Theory of Parsing, Translation, and Compiling》Aho and Ullman, Englewood Cliffs, NJ: Prentice-Hall.）

首先，所有C程序都是由一个或更多的函数加上全局变量（如果存在的话）组成。一个函数是由一个函数名，它的参数表，及与该函数的代码块组成。一个块以"{"开头，后跟一个或多个语句，并以}结束。在C语句中，一个语句可以以C关键字开头（例如if），也可以是一个表达式。（在下一节，我们讨论表达式的构造）总之，我们可写出如下的变换式（有时称为产生式）：

```
program→collection of functions (plus global variables)
function→function-specifier parameter-list code-block
code-block→ { statement sequence }
statement→keyword, expression, or code-block
```

所有C语言程序都以调用main（）开始，以遇到main()中最后一个}或return结束——假设在别处没有调用exit（）或abort（），程序中的其它函数必须直接或间接由main（）调用；因此，执行C语言程序，从main（）的开头开始执行，当main（）结束时，就停止执行。这正是Little C所完成的工作。

·C语言表达式

相对于其它许多计算机语言来说，C扩充了表达式的作用。在C中，一个语句是一个C关键字语句，如While或switch，或者是一个表达式。为了论述方便起见，我们把以C语言关键字开头的所有语句都归类为关键字语句。根据定义，C中的非关键字语句就都是表达式。因此，在C语言中，下述语句均为表达式：

```
count=100;                    /* line 1 */
sample=i/22*(C-10);           /* line 2 */
printf("this is an expression"); /* line 3 */
```

我们再来仔细看看这些表达式语句。在C中，等号是一个赋值运算符。C语言处理赋值运算的方式与BASIC这类语言不同。在BASIC中，等号右边产生的值被赋给左边的变量，但是，重要之处在于：在BASIC中，整个语句并没有值。在C中，等号是赋值运算符，由赋值运算产生的值同表达式右边产生的值是相等的。因此，在C中，赋值语句实质上是赋值表达式；因为它是表达式，所以就有值。这就是我们可以写出下述合法表达式的原因：

```
a=b=c=100;
printf("%d", a= 4 + 5 );
```

在C中这可以行得通的原因是因为赋值是能够产生值的运算。

第2行是一个更复杂的赋值运算。

在第3行中，printf( )用来输出一个字符串。在传统C语言中，无论显式说明与否，所有函数都要返回值，（ANSI标准中加进了void类型允许说明不返回值的函数，但在我们考虑的范围内，这是一种特殊情况，在Litte C中，我们忽略这一点。）因此，一个函数调用是一个可以返回值的表达式——而不管该值是否能被利用。

·表达式求值

在我们编写能正确求C语言表达式的值的代码之前，你必须了解如何用更形式化的术语来定义表达式。实际上在所有计算机语言中，都是用一组产生式规则来递归定义表达式的。Little C解释器支持下列运算：+，-，*，/，%，=，关系运算符(<，==，>等等)，以及括号。因此，我们可以用下述产生或定义Little C的表达式：

```
expression→[assignment][=rvalue]
assignment→lvalue=rvalue
lvalue→variable
rvalue→part[rel-op part]
part→term[+term][-term]
term→factor[*factor][/factor][%factor]
factor→part[rel_op  part]
part→[+ or-] atom
atom→variable, constant, function, or (expression)
```

这里，rel-op指C的关系运算符。lvalue和rvalue分别指赋值语句左边和右边的对象。你应该意识到的一件事是运算符的优先级是隐含在产生式规则中的，优先级越高，运算符出现得越后。

为了观察一下如何用这些规则对表达式求值，我们考虑下面这个例子：

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

首先，用规则1把表达式分成三部分：

```
count          =          10-5*3
  ↑            ↑            ↑
lvalue   assignment operator  rvalue
```

因为在rvalue中没有关系运算等，因此调用term产生式。

```
10        -        5*3
↑         ↑         ↑
term    minus      term
```

当然，第2项（term）是由两个因子（factor）3和5组成的。这两个因子是常数，它们表示最低层产生式规则。然后，沿产生式回溯来计算表达式的值。首先，执行5*3，得到15。然后从10中减去该值，得到-5。最后，把这个值赋给count，这也是整个表达式的值。

开发Little C解释器的第一件事就是开发一个按上述方法对表达式求值的程序。

1.5 表达式语法分析器

读入并分析表达式的代码段称为表达式语法分析器。毫无疑问，表达式语法分析器是Little C解释器中最主要的子系统。由于C语言表达式的范围比其它许多语言的要广，因此表达式语法分析器要有相当的代码量。

可以用多种不同方法来设计C的表达式语法分析器。很多商品化的编译器都使用表格驱动的语法分析器，这些分析器通常都是由语法分析器生成程序产生的。虽然表格驱动的分析器要比其它的运行速度快一些，但用手工很难实现。在Little C解释器中，我们使用递归下降语法分析器，它是按前一节中讨论的产生式规则来实现的。

递归下降分析器基本上是一个彼此递归调用的函数，这些函数分别对表达式进行处理，如果分析器用在编译器中，那么它就生成源代码的目标码。然而，在解释器中，语法分析器的目的是对给定的表达式求值。本章中使用的语法分析器也是基于这一基本点的，以下的大多数论述都是关于如何使用分析器来支持C解释器。

·缩减源代码

所有解释器（编译器）的基础都是一个负责读入源代码，并返回下一个逻辑符号的特殊函数，由于历史原因，这些逻辑符号通常称为符号（tokens），通常的计算机语言，特别是C语言，都用符号（tokens）来定义程序，你可以把一个token当成不可分的程序单元，例如：相等运算符==就是一个token，在不改变其含义条件下，这两个等号是不能分开的。同样，if是一个token，这里的i和f本身对C语言来说，没有任何含义。

在ANSI C的标准中，符号（tokens）被定义成是属于以下这几组的：

关键字（keywords）　标识符（identifiers）　常量（constants）

字符串（strings）　运算符（operators）　标点（punctuation）

关键字是组成C语言的那些tokens，例如while。标识符是变量名、函数名和用户类型名（在Little C中没有实现）。同运算符一样，常量和字符串是自说明的，标点包括一些基本项，如分号，逗号，大括号和圆括号（根据用法不同，有些标点也是运算符）。考虑下面语句：

```
for ( x= 0; x<10; x=x+ 1 ) printf ( "hello %d", x );
```

从左到右，可产生下列符号（tokens）

token	类型
⋮	⋮
for	keyword
(	punctuation
x	identifier
=	operator
0	constant
;	punctuation
x	identifier
<	operator

10	constant
;	punctuation
x	identifier
=	operator
x	identifier
+	operator
1	constant
)	punctuation
printf	identifier
(	punctuation
"hello %d"	string
,	punctuation
x	identifier
)	punctuation
;	punctuation

然而，为方便起见，Little C将tokens分成如下几类：

符号类型	组成成分
分隔符（delimiters）	标点、运算符
关键字（keywords）	关键字
字符串（strings）	引号括起的字符串
标识符（identifiers）	变量和函数名
数字（number）	数字常量
块（block）	{和}

Little C解释器中，返回源代码的符号（tokens）的函数称为get_token（），该函数如下所示：

```
/* Get a token. */
get_token(void)
{

  register char *temp;

  token_type = 0; tok = 0;

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

  /* skip over white space */
  while(iswhite(*prog) && *prog) ++prog;

 if(*prog=='\r') {
   ++prog;
   ++prog;
   /* skip over white space */
   while(iswhite(*prog) && *prog) ++prog;
 }
```

```
  if(*prog=='\0') { /* end of file */
    *token = '\0';
    tok = FINISHED;
    return(token_type=DELIMITER);
  }

  if(strchr("{}", *prog)) { /* block delimiters */
    *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++;
      }


if(strchr("!<>=", *prog)) { /* is or might be
                               a relation operator */
  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)=='=') {
      ppog++; prog++;
      *temp = LE; temp++; *temp = LE;
    }
    else {
      prog++;
      *temp = LT;
    }
    temp++;
    *temp = '\0';
    break;
  case '>': if(*(prog+1)=='=') {
      prog++; prog++;
      *temp = GE; temp++; *temp = GE;
    }
    else {
      prog++;
      *temp = GT;
    }
    temp++;
      *temp = '\0';
      break;
  }
  if(*token) return(token_type = DELIMITER);
}

 f(strchr("+-*^/%=;(),'", *prog)){ /* delimiter */
  *temp = *prog;
  prog++; /* advance to next position */
  temp++;
  *temp = '\0';
  return (token_type=DELIMITER);
}

if(*p  g=='"') { /* quoted string */
  prog++;
  while(*prog!='"'&& *prog!='\r') *temp++ = *prog++;
  if(*prog=='\r') sntx_err(SYNTAX);
  prog++; *temp = '\0';
  return(token_type=STRING);
}

if(isdigit(*prog)) { /* number *,
  while(!isdelim(*prog)) *temp++ = *prog++;
  *temp = '\0';
  return(token_type = NUMBER);
}
```

```
  if(isalpha(*prog)) { /* var or command */
    while(!isdelim(*prog)) *temp++ = *prog++;
    token_type=TEMP;
  }

  *temp = '\0';

  /* see if a string is a command or a variable */
  if(token_type==TEMP) {
    tok = look_up(token); /* convert to internal rep */
    if(tok) token_type = KEYWORD; /* is a keyword */
    else token_type = IDENTIFIER;
  }
  return token_type;
}
```

get__token ( ) 函数使用下列全局数据和枚举类型：

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

char token[80]; /* holds string representation of token */
char token_type; /* contains the type of token */
char tok; /* holds the internal representation of token if
             it is a keyword */

enum tok_types {DELIMITER, IDENTIFIER, NUMBER, KEYWORD, TEMP,
                STRING, BLOCK};

enum double_ops {LT=1, LE, GT  GE, EQ, NE};

/* These are the constants used to call sntx_err() when
   a syntax error occurs.  Add more if you like.
   NOTE: SYNTAX is a generic error message used whe
   nothing else seems appropriate.
*/
enum error_msg
     {SYNTAX, UNBAL_PARENS, NO_EXP, EQUALS_EXPECTED,
      NOT_VAR, PARAM_ERR, SEMI_EXPECTED,
      UNBAL_BRACES, FUNC_UNDEF, TYPE_EXPECTED,
      NEST_FUNC, RET_NOCALL, PARENTEXPECTED,
      WHILE_EXPECTED, QUOTE_EXPECTED, NOT_TEMP,
      TOO_MANY_LVARS};
```

prog指向源代码的当前位置，解释器不改变p__buf指针的值，它总是指向要解释的程序的起始位置。get__token ( ) 函数首先跳过包括回车换行在内的空格符。由于C语言的token不包括空格符（括起的字符串或字符常量除外），因此必须跳过空格符。get__token ( ) 函