Expert C Programming

Deep C Secrets

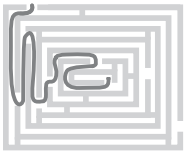
Peter van der Linden

Expert C Programming

Deep C Secrets

Peter van der Linden

Addison-Wesley



Programming Solution

The Piece of Code that Understandeth All Parsing

```
1   #include <stdio.h>
2   #include <string.h>
3   #include <ctype.h>
4   #include <stdlib.h>
5   #define MAXTOKENS 100
6   #define MAXTOKENLEN 64
7
8   enum type_tag { IDENTIFIER, QUALIFIER, TYPE };
9
10  struct token {
11      char type;
12      char string[MAXTOKENLEN];
13  };
14
15  int top=-1;
16  struct token stack[MAXTOKENS];
17  struct token this;
18
19  #define pop stack[top--]
20  #define push(s) stack[++top]=s
21
22  enum type_tag classify_string(void)
23  /* figure out the identifier type */
24  {
25      char *s = this.string;
26      if (!strcmp(s,"const")) {
27          strcpy(s,"read-only");
28          return QUALIFIER;
29      }
30      if (!strcmp(s,"volatile")) return QUALIFIER;
31      if (!strcmp(s,"void")) return TYPE;
32      if (!strcmp(s,"char")) return TYPE;
33      if (!strcmp(s,"signed")) return TYPE;
```

The Piece of Code that Understandeth All Parsing (Continued)

```
34      if (!strcmp(s,"unsigned")) return TYPE;
35      if (!strcmp(s,"short")) return TYPE;
36      if (!strcmp(s,"int")) return TYPE;
37      if (!strcmp(s,"long")) return TYPE;
38      if (!strcmp(s,"float")) return TYPE;
39      if (!strcmp(s,"double")) return TYPE;
40      if (!strcmp(s,"struct")) return TYPE;
41      if (!strcmp(s,"union")) return TYPE;
42      if (!strcmp(s,"enum")) return TYPE;
43      return IDENTIFIER;
44  }
45
46  void gettoken(void) /* read next token into "this" */
47  {
48      char *p = this.string;
49
50      /* read past any spaces */
51      while ((*p = getchar()) == ' ' ) ;
52
53      if (isalnum(*p)) {
54           /* it starts with A-Z,0-9 read in identifier */
55           while ( isalnum(*++p=getchar()) );
56           ungetc(*p,stdin);
57           *p =  '\0';
58           this.type=classify_string();
59           return;
60      }
61
62      if (*p=='*') {
63           strcpy(this.string,"pointer to");
64           this.type = '*';
65           return;
66      }
67      this.string[1]= '\0';
68      this.type = *p;
69      return;
70  }
```

The Piece of Code that Understandeth All Parsing (Continued)

```
/* The piece of code that understandeth all parsing. */
read_to_first_identifier() {
    gettoken();
    while (this.type!=IDENTIFIER) {
        push(this);
        gettoken();
    }
    printf("%s is ", this.string);
    gettoken();
}

deal_with_arrays() {
    while (this.type=='[') {
        printf("array ");
        gettoken(); /* a number or ']' */
        if (isdigit(this.string[0])) {
            printf("0..%d ",atoi(this.string)-1);
            gettoken(); /* read the ']' */
        }
        gettoken(); /* read next past the ']' */
        printf("of ");
    }
}

deal_with_function_args() {
    while (this.type!=')') {
        gettoken();
    }
    gettoken();
    printf("function returning ");
}

deal_with_pointers() {
    while ( stack[top].type== '*' ) {
        printf("%s ", pop.string );
    }
}

```

The Piece of Code that Understandeth All Parsing (Continued)

```
109 deal_with_declarator() {
110     /* deal with possible array/function following the identifier */
111     switch (this.type) {
112     case '[' : deal_with_arrays(); break;
113     case '(' : deal_with_function_args();
114     }
115
116     deal_with_pointers();
117
118     /* process tokens that we stacked while reading to identifier */
119     while (top>=0) {
120         if (stack[top].type == '(' ) {
121             pop;
122             gettoken(); /* read past ')' */
123             deal_with_declarator();
124         } else {
125             printf("%s ",pop.string);
126         }
127     }
128 }
129
130 main()
131 {
132     /* put tokens on stack until we reach identifier */
133     read_to_first_identifier();
134     deal_with_declarator();
135     printf("\n");
136     return 0;
137 }
```

Handy Heuristic

Make String Comparison Look More Natural

One of the problems with the `strcmp ()` routine to compare two strings is that it returns zero if the strings are identical. This leads to convoluted code when the comparison is part of a conditional statement:

```
if (!strcmp(s,"volatile")) return QUALIFIER;
```

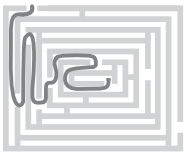
a zero result indicates false, so we have to negate it to get what we want.
Here's a better way. Set up the definition:

```
#define STRCMP(a,R,b)   (strcmp(a,b) R 0)
```

Now you can write a string in the natural style

```
if ( STRCMP(s, ==, "volatile"))
```

Using this definition, the code expresses what is happening in a more natural style. Try rewriting the cdecl program to use this style of string comparison, and see if you prefer it.



Programming Solution

Unscrambling a C Declaration (One More Time)

Here is the solution to "What is this declaration?" on page 78. In each step, the portion of the declaration we are dealing with is printed in bold type. Starting at step one, we will proceed through these steps:

Declaration Remaining	Next Step to Apply	Result
start at the leftmost identifier		
`char *(*c[10])(int **p);`	step 1	say "**c is a…**"
`char *(* [10])(int **p);`	step 2	say "**array[0..9] of…**"

Unscrambling a C Declaration (One More Time)

`char *(*    )(int **p);`	step 5	say "**pointer to…**" go to step 4
`char *(     )(int **p);`	step 4	delete the parens, go to step 2, fall through step 2 to step 3
`char *       (int **p);`	step 3	say "**function returning…**"
`char *              ;`	step 5	say "**pointer to…**"
`char                ;`	step 6	say "**char;**"

Then put it all together to read:

"c is an array[0..9] of pointer to a function returning a pointer-to-char"

and we're done. Note: the fuctions pointed to in the array take a pointer to a pointer as their one and only parameter.