unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-0.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-1.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-2.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-3.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-4.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-5.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-6.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-7.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-8.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-9.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-10.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-11.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-12.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-13.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-14.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-15.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-16.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-17.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-18.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-19.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-20.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-21.jpg

unix C/2349d952-0a52-4a5d-804a-7551a0a07c4c-22.jpg

```
struct tnode space[nwords];                /* the words themselves */
int nnodes nwords;                         /* number of remaining slots */
struct tnode *spacep space;                /* next available slot */
struct tnode *freep;                       /* free list */
/*
 * The main routine reads words until end-of-file ('\0' returned from "getchar")
 * "tree" is called to sort each word into the tree.
 */
main()
{
        struct tnode *top, *tree();
        char c, word[wsize];
        int i;

        i = top = 0;
        while (c=getchar())
                if ('a'<=c && c<='z' || 'A'<=c && c <='Z') {
                        if (i<wsize-1)
                                word[i++] = c;
                } else
                        if (i) {
                                word[i++] = '\0';
                                top = tree(top, word);
                                i = 0;
                        }
        tprint(top);
}
/*
 * The central routine. If the subtree pointer is null, allocate a new node for it.
 * If the new word and the node's word are the same, increase the node's count.
 * Otherwise, recursively sort the word into the left or right subtree according
 * as the argument word is less or greater than the node's word.
 */
struct tnode *tree(p, word)
struct tnode *p;
char word[];
{
        struct tnode *alloc();
        int cond;

        /* Is pointer null? */
        if (p==0) {
                p = alloc();
                copy(word, p->tword);
                p->count = 1;
                p->right = p->left = 0;
                return(p);
        }
        /* Is word repeated? */
        if ((cond=compar(p->tword, word)) == 0) {
                p->count++;
                return(p);
        }
        /* Sort into left or right */
        if (cond<0)
                p->left = tree(p->left, word);
        else
                p->right = tree(p->right, word);
```

```
        return(p);
}
/*
 * Print the tree by printing the left subtree, the given node, and the right subtree.
 */
tprint(p)
struct tnode *p;
{
        while (p) {
                tprint(p->left);
                printf("%d:  %s\n", p->count, p->tword);
                p = p->right;
        }
}
/*
 * String comparison: return number (>, =, <) 0
 * according as s1 (>, =, <) s2
 */
compar(s1, s2)
char *s1, *s2;
{
        int c1, c2;

        while((c1 = *s1++) == (c2 = *s2++))
                if (c1=='\0')
                        return(0);
        return(c2-c1);
}
/*
 * String copy: copy s1 into s2 until the null
 * character appears.
 */
copy(s1, s2)
char *s1, *s2;
{
        while(*s2++ = *s1++);
}
/*
 * Node allocation: return pointer to a free node.
 * Bomb out when all are gone.  Just for fun, there
 * is a mechanism for using nodes that have been
 * freed, even though no one here calls "free."
 */
struct tnode *alloc()
{
        struct tnode *t;

        if (freep) {
                t = freep;
                freep = freep->left;
                return(t);
        }
        if (--nnodes < 0) {
                printf("Out of space\n");
                exit();
        }
        return(spacep++);
}
/*
```

```
	* The uncalled routine which puts a node on the free list.
	*/
	free(p)
	struct tnode *p;
	{
		p->left = freep;
		freep = p;
	}
```

To illustrate a slightly different technique of handling the same problem, we will repeat fragments of this example with the tree nodes treated explicitly as members of an array. The fundamental change is to deal with the subscript of the array member under discussion, instead of a pointer to it. The struct declaration becomes

```
	struct tnode {
		char tword[wsize];
		int count;
		int left;
		int right;
	};
```

and *alloc* becomes

```
	alloc()
	{
		int t;

		t = --nnodes;
		if (t<=0) {
			printf("Out of space\n");
			exit();
		}
		return(t);
	}
```

The *free* stuff has disappeared because if we deal with exclusively with subscripts some sort of map has to be kept, which is too much trouble.

Now the *tree* routine returns a subscript also, and it becomes:

```
	tree(p, word)
	char word[];
	{
		int cond;

		if (p==0) {
			p = alloc();
			copy(word, space[p].tword);
			space[p].count = 1;
			space[p].right = space[p].left = 0;
			return(p);
		}
		if ((cond=compar(space[p].tword, word)) == 0) {
			space[p].count++;
			return(p);
		}
		if (cond<0)
			space[p].left = tree(space[p].left, word);
		else
			space[p].right = tree(space[p].right, word);
		return(p);
	}
```

The other routines are changed similarly. It must be pointed out that this version is noticeably less efficient than the first because of the multiplications which must be done to compute an offset in *space* corresponding to the subscripts.

The observation that subscripts (like "a[i]") are less efficient than pointer indirection (like "*ap") holds true independently of whether or not structures are involved. There are of course many situations where subscripts are indispensable, and others where the loss in efficiency is worth a gain in clarity.

16.3 Formatted output

Here is a simplified version of the *printf* routine, which is available in the C library. It accepts a string (character array) as first argument, and prints subsequent arguments according to specifications contained in this format string. Most characters in the string are simply copied to the output; two-character sequences beginning with "%" specify that the next argument should be printed in a style as follows:

%d	decimal number
%o	octal number
%c	ASCII character, or 2 characters if upper character is not null
%s	string (null-terminated array of characters)
%f	floating-point number

The actual parameters for each function call are laid out contiguously in increasing storage locations; therefore, a function with a variable number of arguments may take the address of (say) its first argument, and access the remaining arguments by use of subscripting (regarding the arguments as an array) or by indirection combined with pointer incrementation.

If in such a situation the arguments have mixed types, or if in general one wishes to insist that an lvalue should be treated as having a given type, then struct declarations like those illustrated below will be useful. It should be evident, though, that such techniques are implementation dependent.

Printf depends as well on the fact that char and float arguments are widened respectively to int and double, so there are effectively only two sizes of arguments to deal with. *Printf* calls the library routines *putchar* to write out single characters and *ftoa* to dispose of floating-point numbers.

```
printf(fmt, args)
char fmt[ ];
{
        char *s;
        struct { char **charpp; };
        struct { double *doublep; };
        int *ap, x, c;

        ap = &args;                     /* argument pointer */
        for ( ; ; ) {
                while((c = *fmt++) != '%') {
                        if(c == '\0')
                                return;
                        putchar(c);
                }
                switch (c = *fmt++) {
                /* decimal */
                case 'd':
                        x = *ap++;
                        if(x < 0) {
                                x = -x;
                                if(x<0) {       /* is - infinity */
                                        printf("-32768");
                                        continue;
                                }
```

```
                                putchar('-');
                        }
                        printd(x);
                        continue;
                /* octal */
                case 'o':
                        printo(*ap++);
                        continue;
                /* float, double */
                case 'f':
                        /* let ftoa do the real work */
                        ftoa(*ap.doublep++);
                        continue;
                /* character */
                case 'c':
                        putchar(*ap++);
                        continue;
                /* string */
                case 's':
                        s = *ap.charpp++;
                        while(c = *s++)
                                putchar(c);
                        continue;
                }
                putchar(c);
        }
}
/*
 * Print n in decimal; n must be non-negative
 */
printd(n)
{
        int a;
        if (a=n/10)
                printd(a);
        putchar(n%10 + '0');
}
/*
 * Print n in octal, with exactly 1 leading 0
 */
printo(n)
{
        if (n)
                printo((n>>3)&017777);
        putchar((n&07)+'0');
}
```

MH-1273-DMR-   D. M. Ritchie

Att:
References
Appendix 1

REFERENCES

1. Johnson, S. C., and Kernighan, B. W. "The Programming Language B." Comp. Sci. Tech. Rep. #8., Bell Laboratories, 1972.
2. Ritchie, D. M., and Thompson, K. L. "The UNIX Time-sharing System." C. ACM 7, 17, July, 1974, pp. 365-375.
3. Peterson, T. G., and Lesk, M. E. "A User's Guide to the C Language on the IBM 370." Internal Memorandum, Bell Laboratories, 1974.
4. Thompson, K. L., and Ritchie, D. M. *UNIX Programmer's Manual.* Bell Laboratories, 1973.
5. Lesk, M. E., and Barres, B. A. "The GCOS C Library." Internal memorandum, Bell Laboratories, 1974.
6. Kernighan, B. W. "Programming in C– A Tutorial." Unpublished internal memorandum, Bell Laboratories, 1974.

APPENDIX 1
Syntax Summary

1. Expressions.

```
expression:
        primary
        * expression
        & expression
        - expression
        ! expression
        ~ expression
        ++ lvalue
        -- lvalue
        lvalue ++
        lvalue --
        sizeof expression
        expression binop expression
        expression ? expression : expression
        lvalue asgnop expression
        expression , expression

primary:
        identifier
        constant
        string
        ( expression )
        primary ( expression-list_opt )
        primary [ expression ]
        lvalue . identifier
        primary -> identifier

lvalue:
        identifier
        primary [ expression ]
        lvalue . identifier
        primary -> identifier
        * expression
        ( lvalue )
```

The primary-expression operators

```
() [] . ->
```

have highest priority and group left-to-right. The unary operators

```
& - ! ~ ++ -- sizeof
```

have priority below the primary operators but higher than any binary operator, and group right-to-left. Binary operators and the conditional operator all group left-to-right, and have priority decreasing as indicated:

```
binop:
        *   /   %
        +   -
        >>   <<
        <   >   <=   >=
        ==   !=
```

&
^
|
&&
||
? :

Assignment operators all have the same priority, and all group right-to-left.

> *asgnop:*
> = =+ =- =* =/ =% =>> =<< =& =^ =|

The comma operator has the lowest priority, and groups left-to-right.

2. Declarations.

> *declaration:*
> *decl-specifiers declarator-list*$_{opt}$;

> *decl-specifiers:*
> *type-specifier*
> *sc-specifier*
> *type-specifier sc-specifier*
> *sc-specifier type-specifier*

> *sc-specifier:*
> auto
> static
> extern
> register

> *type-specifier:*
> int
> char
> float
> double
> struct { *type-decl-list* }
> struct *identifier* { *type-decl-list* }
> struct *identifier*

> *declarator-list:*
> *declarator*
> *declarator* , *declarator-list*

> *declarator:*
> *identifier*
> * *declarator*
> *declarator* ( )
> *declarator* [*constant-expression*$_{opt}$]
> (*declarator*)

> *type-decl-list:*
> *type-declaration*
> *type-declaration type-decl-list*

> *type-declaration:*
> *type-specifier declarator-list* ;

3. Statements.

> *statement:*
> *expression* ;

{ *statement-list* }
if (*expression*) *statement*
if (*expression*) *statement* else *statement*
while (*expression*) *statement*
for (*expression*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$) statement
switch (*expression*) *statement*
case *constant-expression* : *statement*
default : *statement*
break ;
continue ;
return ;
return (*expression*) ;
goto *expression* ;
identifier : *statement*
;

statement-list:
statement
statement statement-list

4. External definitions.

program:
external-definition
external-definition program

external-definition:
function-definition
data-definition

function-definition:
type-specifier$_{opt}$ *function-declarator function-body*

function-declarator:
declarator (*parameter-list*$_{opt}$)

parameter-list:
identifier
identifier , *parameter-list*

function-body:
type-decl-list function-statement

function-statement:
{ *declaration-list*$_{opt}$ *statement-list* }

data-definition:
extern$_{opt}$ *type-specifier*$_{opt}$ *init-declarator-list*$_{opt}$;

init-declarator-list:
init-declarator
init-declarator , *init-declarator-list*

init-declarator:
declarator initializer$_{opt}$

initializer:
  constant
  { *constant-expression-list* }

constant-expression-list:
  constant-expression
  constant-expression , *constant-expression-list*

constant-expression:
  expression

5. Preprocessor

define *identifier token-string*

include "*filename*"