CHAPTER 6: STRUCTURES

A *structure* is a collection of one or more variables, possibly of different   
types, grouped together under a single name for convenient handling.   
(Structures are called "records" in some languages, most notably Pascal.)

The traditional example of a structure is the payroll record: an   
"employee" is described by a set of attributes such as name, address, social   
security number, salary, etc. Some of these in turn could be structures: a   
name has several components, as does an address and even a salary.

Structures help to organize complicated data, particularly in large pro­   
grams, because in many situations they permit a group of related variables   
to be treated as a unit instead of as separate entities. In this chapter we will   
try to illustrate how structures are used. The programs we will use are   
bigger than many of the others in the book, but still of modest size.

6.1 Basics

Let us revisit the date conversion routines of Chapter 5. A date consists   
of several parts, such as the day, month, and year, and perhaps the day of   
the year and the month name. These five variables can all be placed into a   
single structure like this:

struct date (

int day;

int month;

int year;

int yearday;

char mon\_name [4] ;

The keyword struct introduces a structure declaration, which is a list   
of declarations enclosed in braces. An optional name called a *structure tag*    
may follow the word struct (as with date here). The tag names this   
kind of structure, and can be used subsequently as a shorthand for the   
detailed declaration.

119

120 THE C PROGRAMMING LANGUAGE CHAPTER 6

The elements or variables mentioned in a structure are called *members.*    
A structure member or tag and an ordinary (i.e., non-member) variable can   
have the same name without conflict, since they can always be distinguished   
by context. Of course as a matter of style one would normally use the same   
names only for closely related objects.

The right brace that terminates the list of members may be followed by   
a list of variables, just as for any basic type. That is,

struct I ... I x, y, z;

is syntactically analogous to

int x, y, z;

in the sense that each statement declares x, y and z to be variables of the   
named type and causes space to be allocated for them.

A structure declaration that is not followed by a list of variables allocates   
no storage; it merely describes a *template* or the shape of a structure. If the   
declaration is tagged, however, the tag can be used later in definitions of   
actual instances of the structure. For example, given the declaration of   
date above,

struct date d;

defines a variable d which is a structure of type date. An external or static   
structure can be initialized by following its definition with a list of initializers   
for the components:

struct date d =1 4, 7, 1776, 186, "Jul" );

A member of a particular structure is referred to in an expression by a   
construction of the form

*structure-name. member*

The structure member operator " " connects the structure name and the   
member name. To set leap from the date in structure d, for example,

leap = d.year % 4 == 0 && d.year % 100 != 0   
II d.year % 400 == 0;

or to check the month name,

if (strcmp(d.mon\_name, "Aug") == 0) ...

or to convert the first character of the month name to lower case,

d.mon\_name[0] = lower(d.mon\_name[0]);

Structures can be nested; a payroll record might actually look like

CHAPTER6 STRUCTURES 121

struct person (

char name[NAMESIZE];

char address[ADRSIZE];

long zipcode;

long ss\_number;

double salary;

struct date birthdate;

struct date hiredate;

The person structure contains two dates. If we declare emp as

struct person emp;

then

emp.birthdate.month

refers to the month of birth. The structure member operator . associates   
left to right.

6.2 Structures and Functions

There are a number of restrictions on C structures. The essential rules   
are that the only operations that you can perform on a structure are take its   
address with &, and access one of its members. This implies that structures   
may not be assigned to or copied as a unit, and that they can not be passed   
to or returned from functions. (These restrictions will be removed in forth­   
coming versions.) Pointers to structures do not suffer these limitations,   
however, so structures and functions do work together comfortably. Finally,   
automatic structures, like automatic arrays, cannot be initialized; only exter­   
nal or static structures can.

Let us investigate some of these points by rewriting the date conversion   
functions of the last chapter to use structures. Since the rules prohibit pass­   
ing a structure to a function directly, we must either pass the components   
separately, or pass a pointer to the whole thing. The first alternative uses   
day\_of\_year as we wrote it in Chapter 5:

d.yearday = day\_of\_year(d.year, d.month, d.day);

The other way is to pass a pointer. If we have declared hiredate as

struct date hiredate;

and re-written day\_of\_year, we can then say

hiredate.yearday = day\_of\_year(&hiredate);

to pass a pointer to hiredate to day\_of\_year. The function has to be   
modified because its argument is now a pointer rather than a list of vari­   
ables.

122 THE C PROGRAMMING LANGUAGE CHAPTER 6

day\_of\_year(pd) /\* set day of year from month, day \*/   
struct date \*pd;

int i, day, leap;

day = pd->day;

leap = pd->year % 4 == 0 && pd->year % 100 != 0

II pd->year % 400 == 0;

for (i = 1; i < pd->month; i++)

day += day\_tab[leap][i];

return (day);

The declaration

struct date \*pd;

says that pd is a pointer to a structure of type date. The notation   
exemplified by

pd->year

is new. If p is a pointer to a structure, then

*p->member-of-structure*

refers to the particular member. (The operator —> is a minus sign followed   
by >.)

Since pd points to the structure, the year member could also be   
referred to as

(\*pd).year

but pointers to structures are so frequently used that the —> notation is pro­   
vided as a convenient shorthand. The parentheses are necessary in   
(\*pd) .year because the precedence of the structure member operator . is   
higher than \*. Both —> and . associate from left to right, so

p->q->memb

emp.birthdate.month

are

(p->q)->memb

(emp.birthdate).month

For completeness here is the other function, month\_day, rewritten to   
use the structure.

CHAPTER 6 STRUCTURES 123

month\_day(pd) /\* set month and day from day of year \*/   
struct date \*pd;

int i, leap;

leap = pd->year % 4 == 0 && pd->year % 100 != 0

II pd->year % 400 == 0;

pd->day = pd->yearday;

for (i = 1; pd->day > day\_tab[leap][i]; i++)

pd->day -= day\_tab[leap][i];

pd->month = i;

The structure operators —> and . , together with ( ) for argument lists   
and [ ] for subscripts, are at the top of the precedence hierarchy and thus   
bind very tightly. For example, given the declaration

struct (

int x;   
int \*Y;

) \*P;

then

++p->x

increments x, not p, because the implied parenthesization is ++ (p—>x ) .   
Parentheses can be used to alter the binding: (++p) —>x increments p   
before accessing x, and (p++)—>x increments p afterward. (This last set   
of parentheses is unnecessary. Why?)

In the same way, \*p—>y fetches whatever y points to; \*p—>y++ incre­   
ments y after accessing whatever it points to (just like \*s++); (\*p—>y) ++   
increments whatever y points to; and \*p++—>y increments p after accessing   
whatever y points to.

6.3 Arrays of Structures

Structures are especially suitable for managing arrays of related vari­   
ables. For instance, consider a program to count the occurrences of each C   
keyword. We need an array of character strings to hold the names, and an   
array of integers for the counts. One possibility is to use two parallel arrays   
keyword and keycount, as in

char \*keyword[NKEYS];   
int keycount[NKEYS];

But the very fact that the arrays are parallel indicates that a different organi­   
zation is possible. Each keyword entry is really a pair:

124 THE C PROGRAMMING LANGUAGE CHAPTER 6

char \*keyword;   
int keycount;

and there is an array of pairs. The structure declaration

struct key (

char \*keyword;   
int keycount;   
) keytab[NKEYS);

defines an array keytab of structures of this type, and allocates storage to   
them. Each element of the array is a structure. This could also be written

struct key (

char \*keyword;

int keycount;

) ;

struct key keytab[NKEYS];

Since the structure keytab actually contains a constant set of names, it   
is easiest to initialize it once and for all when it is defined. The structure   
initialization is quite analogous to earlier ones — the definition is followed   
by a list of initializers enclosed in braces:

struct key (

char \*keyword;   
int keycount;   
) keytab[l =(

"break", 0,

"case", 0,

"char", 0,

"continue", 0,   
"default", 0,

/\* ... \*/

"unsigned", 0,   
"while", 0

} ;

The initializers are listed in pairs corresponding to the structure members.   
It would be more precise to enclose initializers for each "row" or structure   
in braces, as in

( "break", 0 ),   
{ "case", 0 1,

* • •

but the inner braces are not necessary when the initializers are simple vari­   
ables or character strings, and when all are present. As usual, the compiler   
will compute the number of entries in the array keytab if initializers are   
present and the [1 is left empty.

CHAPTER 6 STRUCTURES 125

The keyword-counting program begins with the definition of keytab.   
The main routine reads the input by repeatedly calling a function getword   
that fetches the input one word at a time. Each word is looked up in   
keytab with a version of the binary search function that we wrote in   
Chapter 3. (Of course the list of keywords has to be given in increasing   
order for this to work.)

#define MAXWORD 20

main() /\* count C keywords \*/

int n, t;

char word[MAXWORD];

while ((t = getword (word, MAXWORD)) != EOF)

if (t == LETTER)

if ((n = binary(word, keytab, NKEYS)) >= 0)

keytab[n].keycount++;

for (n = 0; n < NKEYS; n++)

if (keytab[n].keycount > 0)

printf("%4d %s\n",

keytab[n].keycount, keytab[n].keyword);

binary(word, tab, n) /\* find word in tab[0]...tab[n-1] \*/

char \*word;

struct key tab[];

int n;

int low, high, mid, cond;

low = 0;

high = n - 1;

while (low <= high) (

mid = (low+high) / 2;

if ((cond = strcmp(word, tab[mid].keyword)) < 0)

high = mid - 1;

else if (cond > 0)

low = mid + 1;

else

return (mid);

return(-1);

We will show the function getword in a moment; for now it suffices to say   
that it returns LETTER each time it finds a word, and copies the word into   
its first argument.

126 THE C PROGRAMMING LANGUAGE CHAPTER 6

The quantity NKEYS is the number of keywords in keytab. Although   
we could count this by hand, it's a lot easier and safer to do it by machine,   
especially if the list is subject to change. One possibility would be to ter­   
minate the list of initializers with a null pointer, then loop along keytab   
until the end is found.

But this is more than is needed, since the size of the array is completely   
determined at compile time. The number of entries is just

*size of* keytab */ size of* struct key

C provides a compile-time unary operator called sizeof which can be used   
to compute the size of any object. The expression

sizeof *(object)*

yields an integer equal to the size of the specified object. (The size is given   
in unspecified units called "bytes," which are the same size as a char.)   
The object can be an actual variable or array or structure, or the name of a   
basic type like int or double, or the name of a derived type like a struc­   
ture. In our case, the number of keywords is the array size divided by the   
size of one array element. This computation is used in a #define state­   
ment to set the value of NKEYS:

#define NKEYS (sizeof(keytab) / sizeof(struct key))

Now for the function getword. We have actually written a more gen­   
eral getword than is necessary for this program, but it is not really much   
more complicated. getword returns the next "word" from the input,   
where a word is either a string of letters and digits beginning with a letter,   
or a single character. The type of the object is returned as a function value;   
it is LETTER if the token is a word, EOF for end of file, or the character   
itself if it is non-alphabetic.

CHAPTER6 STRUCTURES 127

getword(w, lim) /\* get next word from input \*/

char \*w;

int lim;

int c, t;

if (type(c = \*w++ = getch()) != LETTER) (

\*w = '\0';

return(c);

)

while (--lim > 0) (

t = type(c = \*w++ = getch());

if (t != LETTER && t != DIGIT) (

ungetch(c);

break;

)

\*(w-1) = '\0';

return (LETTER);

getword uses the routines getch and ungetch which we wrote in   
Chapter 4: when the collection of an alphabetic token stops, getword has   
gone one character too far. The call to ungetch pushes that character back   
on the input for the next call.

getword calls type to determine the type of each individual character   
of input. Here is a version *.for the ASCII alphabet only.*

type(c) /\* return type of ASCII character \*/   
int c;

if (c >= 'a' && c <= 'z' II c >= 'A' && c <= 'Z')

return (LETTER);

else if (c >= '0' && c <= '9')

return (DIGIT);

else

return(c);

The symbolic constants LETTER and DIGIT can have any values that do   
not conflict with non-alphanumeric characters and EOF; the obvious choices   
are

#define LETTER 'a'   
#define DIGIT '0'

getword can be faster if calls to the function type are replaced by   
references to an appropriate array type []. The standard C library provides   
macros called isalpha and isdigit which operate in this manner.

128 THE C PROGRAMMING LANGUAGE CHAPTER 6

Exercise 6-1. Make this modification to getword and measure the change   
in speed of the program. 0

Exercise 6-2. Write a version of type which is independent of character   
set. 0

Exercise 6-3. Write a version of the keyword-counting program which does   
not count occurrences contained within quoted strings. 0

6.4 Pointers to Structures

To illustrate some of the considerations involved with pointers and   
arrays of structures, let us write the keyword-counting program again, this   
time using pointers instead of array indices.

The external declaration of keytab need not change, but main and   
binary do need modification.

main() /\* count C keywords; pointer version \*/

int t;

char word[MAXWORD];

struct key \*binary(), \*p;

while ((t = getword(word, MAXWORD)) != EOF)

if (t == LETTER)

if ((p=binary(word, keytab, NKEYS)) != NULL)

p->keycount++;

for (p = keytab; p < keytab + NKEYS; p++)

if (p->keycount > 0)

printf("%4d %s\n", p->keycount, p->keyword);

CHAPTER6 STRUCTURES 129

struct key \*binary(word, tab, n) /\* find word \*/

char \*word; /\* in tab[0]...tab[n-1] \*/

struct key tab[];

int n;

int cond;

struct key \*low = &tab[0];   
struct key \*high = &tab[n-1];   
struct key \*mid;

while (low <= high) (

mid = low + (high-low) / 2;

if ((cond = strcmp(word, mid->keyword)) < 0)

high = mid - 1;

else if (cond > 0)

low = mid + 1;

else

return (mid);

return (NULL)

There are several things worthy of note here. First, the declaration of   
binary must indicate that it returns a pointer to the structure type key,   
instead of an integer; this is declared both in main and in binary. If   
binary finds the word, it returns a pointer to it; if it fails, it returns NULL.

Second, all the accessing of elements of keytab is done by pointers.   
This causes one significant change in binary: the computation of the mid­   
dle element can no longer be simply

mid = (low+high) / 2

because the *addition* of two pointers will not produce any kind of a useful   
answer (even when divided by 2), and in fact is illegal. This must be   
changed to

mid = low + (high-low) / 2

which sets mid to point to the element halfway between low and high.

You should also study the initializers for low and high. It is possible   
to initialize a pointer to the address of a previously defined object; that is   
precisely what we have done here.

In main we wrote

for (p = keytab; p < keytab + NKEYS; p++)

If p is a pointer to a structure, any arithmetic on p takes into account the   
actual size of the structure, so p++ increments p by the correct amount to

130 THE C PROGRAMMING LANGUAGE CHAPTER 6

get the next element of the array of structures. But don't assume that the   
size of a structure is the sum of the sizes of its members — because of   
alignment requirements for different objects, there may be "holes" in a   
structure.

Finally, an aside on program format. When a function returns a compli­   
cated type, as in

struct key \*binary(word, tab, n)

the function name can be hard to see, and to find with a text editor.   
Accordingly an alternate style is sometimes used:

struct key \*

binary(word, tab, n)

This is mostly a matter of personal taste; pick the form you like and hold to   
it.

6.5 Self-referential Structures

Suppose we want to handle the more general problem of counting the   
occurrences of *all* the words in some input. Since the list of words isn't   
known in advance, we can't conveniently sort it and use a binary search.   
Yet we can't do a linear search for each word as it arrives, to see if it's   
already been seen; the program would take forever. (More precisely, its   
expected running time would grow quadratically with the number of input   
words.) How can we organize the data to cope efficiently with a list of arbi­   
trary words?

One solution is to keep the set of words seen so far sorted at all times,   
by placing each word into its proper position in the order as it arrives. This   
shouldn't be done by shifting words in a linear array, though — that also   
takes too long. Instead we will use a data structure called a *binary tree.*

The tree contains one "node" per distinct word; each node contains

*a pointer to the text of the word*

*a count of the number of occurrences*

*a pointer to the left child node*

*a pointer to the right child node*

No node may have more than two children; it might have only zero or one.

The nodes are maintained so that at any node the left subtree contains   
only words which are less than the word at the node, and the right subtree   
contains only words that are greater. To find out whether a new word is   
already in the tree, one starts at the root and compares the new word to the   
word stored at that node. If they match, the question is answered   
affirmatively. If the new word is less than the tree word, the search contin­   
ues at the left child; otherwise the right child is investigated. If there is no   
child in the required direction, the new word is not in the tree, and in fact

CHAPTER 6 STRUCTURES 131

the proper place for it to be is the missing child. This search process is   
inherently recursive, since the search from any node uses a search from one   
of its children. Accordingly recursive routines for insertion and printing will   
be most natural.

Going back to the description of a node, it is clearly a structure with   
four components:

struct tnode ( /\* the basic node \*/

char \*word; /\* points to the text \*/

int count; /\* number of occurrences \*/

struct tnode \*left; /\* left child \*/

struct tnode \*right; /\* right child \*/   
;

This "recursive" declaration of a node might look chancy, but it's actually   
quite correct. It is illegal for a structure to contain an instance of itself, but

struct tnode \*left;

declares left to be a *pointer* to a node, not a node itself.

The code for the whole program is surprisingly small, given a handful of   
supporting routines that we have already written. These are getword, to   
fetch each input word, and alloc, to provide space for squirreling the   
words away.

The main routine simply reads words with getword and installs them   
in the tree with tree.

#delime\_ MAXWORD 20

main() /\* word frequency count \*/

|  |  |
| --- | --- |
| struct tnode \*root, \*tree();  char word [MAXWORD];  int t;  root = NULL;  while ((t = getword(word, MAXWORD))  if (t == LETTER)  root = tree(root, word);  treeprint(root); | 1= EOF) |

tree itself is straightforward. A word is presented by main to the top   
level (the root) of the tree. At each stage, that word is compared to the   
word already stored at the node, and is percolated down to either the left or   
right subtree by a recursive call to tree. Eventually the word either   
matches something already in the tree (in which case the count is incre­   
mented), or a null pointer is encountered, indicating that a node must be

132 THE C PROGRAMMING LANGUAGE CHAPTER 6

created and added to the tree. If a new node is created, tree returns a   
pointer to it, which is installed in the parent node.

struct tnode \*tree(p, w) /\* install w at or below P \*/

struct tnode \*p;

char \*w;

(

struct tnode \*talloc();

char \*strsave();

int cond;

if (p == NULL) ( /\* a new word has arrived \*/

p = talloc(); /\* make a new node \*/

p->word = strsave(w);

p->count = 1;

p->left = p->right = NULL;

) else if ((cond = strcmp(w, p->word)) == 0)

p->count++; /\* repeated word \*/

else if (cond < 0) /\* lower goes into left subtree \*/

p->left = tree(p->left, w);

else /\* greater into right subtree \*/

p->right = tree(p->right, w);

return (p);

)

Storage for the new node is fetched by a routine talloc, which is an   
adaptation of the alloc we wrote earlier. It returns a pointer to a free   
space suitable for holding a tree node. (We will discuss this more in a   
moment.) The new word is copied to a hidden place by strsave, the count   
is initialized, and the two children are made null. This part of the code is   
executed only at the edge of the tree, when a new node is being added. We   
have (unwisely for a production program) omitted error checking on the   
values returned by strsave and talloc.

treeprint prints the tree in left subtree order; at each node, it prints   
the left subtree (all the words less than this word), then the word itself,   
then the right subtree (all the words greater). If you feel shaky about recur­   
sion, draw yourself a tree and print it with treeprint; it's one of the   
cleanest recursive routines you can find.

CHAPTER 6 STRUCTURES 133

treeprint(p) /\* print tree p recursively \*/   
struct tnode \*p;

if (p != NULL) (

treeprint(p->left);

printf("%4d %s\n", p->count, p->word);

treeprint(p->right);

A practical note: if the tree becomes "unbalanced" because the words   
don't arrive in random order, the running time of the program can grow too   
fast. As a worst case, if the words are already in order, this program does   
an expensive simulation of linear search. There are generalizations of the   
binary tree, notably 2-3 trees and AVL trees, which do not suffer from this   
worst-case behavior, but we will not describe them here.

Before we leave this example, it is also worth a brief digression on a   
problem related to storage allocators. Clearly it's desirable that there be   
only one storage allocator in a program, even though it allocates different   
kinds of objects. But if one allocator is to process requests for, say, pointers   
to char's and pointers to struct tnode's, two questions arise. First,   
how does it meet the requirement of most real machines that objects of cer­   
tain types must satisfy alignment restrictions (for example, integers often   
must be located on even addresses)? Second, what declarations can cope   
with the fact that alloc necessarily returns different kinds of pointers?

Alignment requirements can generally be satisfied easily, at the cost of   
some wasted space, merely by ensuring that the allocator always returns a   
pointer that meets *all* alignment restrictions. For example, on the PDP-11 it   
is sufficient that alloc always return an even pointer, since any type of   
object may be stored at an even address. The only cost is a wasted character   
on odd-length requests. Similar actions are taken on other machines. Thus   
the implementation of alloc may not be portable, but the usage is. The   
alloc of Chapter 5 does not guarantee any particular alignment; in Chapter   
8 we will show how to do the job right.

The question of the type declaration for alloc is a vexing one for any   
language that takes its type-checking seriously. In C, the best procedure is   
to declare that alloc returns a pointer to char, then explicitly coerce the   
pointer into the desired type with a cast. That is, if p is declared as

char \*p;

then

(struct tnode \*) p

converts it into a tnode pointer in an expression. Thus talloc is written

134 THE C PROGRAMMING LANGUAGE CHAPTER 6

as

ttruct tnode \*tallod()

char \*alloc();

return((struct tnode \*) alloc(sizeof(struct tnode)));

This is more than is needed for current compilers, but represents the safest   
course for the future.

Exercise 6-4. Write a program which reads a C program and prints in alpha­   
betical order each group of variable names which are identical in the first 7   
characters, but different somewhere thereafter. (Make sure that 7 is a   
parameter). 0

Exercise 6-5. Write a basic cross-referencer: a program which prints a list of   
all words in a document, and, for each word, a list of the line numbers on   
which it occurs. El

Exercise 6-6. Write a program which prints the distinct words in its input   
sorted into decreasing order of frequency of occurrence. Precede each word   
by its count. LI

6.6 Table Lookup

In this section we will write the innards of a table-lookup package as an   
illustration of more aspects of structures. This code is typical of what might   
be found in the symbol table management routines of a macro processor or   
a compiler. For example, consider the C #define statement. When a line   
like

#define YES 1

is encountered, the name YES and the replacement text 1 are stored in a   
table. Later, when the name YES appears in a statement like

inword = YES;

it must be replaced by 1.

There are two major routines that manipulate the names and replace­   
ment texts. install (s, t) records the name s and the replacement text   
t in a table; s and t are just character strings. lookup (s) searches for s   
in the table, and returns a pointer to the place where it was found, or NULL   
if it wasn't there.

The algorithm used is a hash search — the incoming name is converted   
into a small positive integer, which is then used to index into an array of   
pointers. An array element points to the beginning of a chain of blocks

CHAPTER 6 STRUCTURES 135

describing names that have that hash value. It is NULL if no names have   
hashed to that value.

A block in the chain is a structure containing pointers to the name, the   
replacement text, and the next block in the chain. A null next-pointer   
marks the end of the chain.

struct nlist ( /\* basic table entry \*/

char \*name;

char \*def;

struct nlist \*next; /\* next entry in chain \*/

;

The pointer array is just

#define HASHSIZE 100

static struct nlist \*hashtab[HASHSIZE]; /\* pointer table \*/

The hashing function, which is used by both lookup and install,   
simply adds up the character values in the string and forms the remainder   
modulo the array size. (This is not the best possible algorithm, but it has   
the merit of extreme simplicity.)

hash(s) /\* form hash value for string s \*/

char \*s;

int hashval;

for (hashval = 0; \*s !=   
hashval += \*s++;

return(hashval % HASHSIZE);

The hashing process produces a starting index in the array hashtab; if   
the string is to be found anywhere, it will be in the chain of blocks begin­   
ning there. The search is performed by lookup. If lookup finds the   
entry already present, it returns a pointer to it; if not, it returns NULL.

struct nlist \*lookup(s) /\* look for s in hashtab \*/   
char \*s;

struct nlist \*np;

for (np = hashtab[hash(s)]; np != NULL; np = np->next)   
if (strcmp(s, np->name) == 0)

return(np); /\* found it \*/   
return(NULL); /\* not found \*/

install uses lookup to determine whether the name being installed   
is already present; if so, the new definition must supersede the old one.

136 THE C PROGRAMMING LANGUAGE CHAPTER 6

Otherwise, a completely new entry is created. install returns NULL if for   
any reason there is no room for a new entry.

struct nlist \*install(name, def) /\* put (name, def) \*/

char \*name, \*def; /\* in hashtab \*/

(

struct nlist \*np, \*lookup();

char \*strsave(), \*alloc();

int hashval;

if ((np = lookup (name)) == NULL) ( /\* not found \*/

np = (struct nlist \*) alloc(sizeof(\*np));

if (np == NULL)

return(NULL);

if ((np->name = strsave(name)) == NULL)

return (NULL)

hashval = hash(np->name);

np->next = hashtab[hashvall;

hashtab[hashval] = np;

) else /\* already there \*/

free(np->def); /\* free previous definition \*/

if ((np->def = strsave(def)) == NULL)

return(NULL);

return(np);

I

strsave merely copies the string given by its argument into a safe   
place, obtained by a call on alloc. We showed the code in Chapter 5.   
Since calls to alloc and free may occur in any order, and since alignment   
matters, the simple version of alloc in Chapter 5 is not adequate here; see   
Chapters 7 and 8.

Exercise 6-7. Write a routine which will remove a name and definition from   
the table maintained by lookup and install. El

Exercise 6-8. Implement a simple version of the #define processor suit­   
able for use with C programs, based on the routines of this section. You   
may also find getch and ungetch helpful. 0

6.7 Fields

When storage space is at a premium, it may be necessary to pack several   
objects into a single machine word; one especially common use is a set of   
single-bit flags in applications like compiler symbol tables. Externally-   
imposed data formats, such as interfaces to hardware devices, also often   
require the ability to get at pieces of a word.

CHAPTER 6 STRUCTURES 137

Imagine a fragment of a compiler that manipulates a symbol table. Each   
identifier in a program has certain information associated with it, for exam-   
pie, whether or not it is a keyword, whether or not it is external and/or   
static, and so on. The most compact way to encode such information is a   
set of one-bit flags in a single char or int.

The usual way this is done is to define a set of "masks" corresponding   
to the relevant bit positions, as in

#define KEYWORD 01   
#define EXTERNAL 02   
#define STATIC 04

(The numbers must be powers of two.) Then accessing the bits becomes a   
matter of "bit-fiddling" with the shifting, masking, and complementing   
operators which were described in Chapter 2.

Certain idioms appear frequently:

flags I= EXTERNAL I STATIC;

turns on the EXTERNAL and STATIC bits in flags, while

flags &= -(EXTERNAL I STATIC);

turns them off, and

if ((flags & (EXTERNAL I STATIC)) == 0) ..

is true if both bits are off.

Although these idioms are readily mastered, as an alternative, C offers   
the capability of defining and accessing fields within a word directly rather   
than by bitwise logical operators. A *field* is a set of adjacent bits within a   
single int. The syntax of field definition and access is based on structures.   
For example, the symbol table #define's above could be replaced by the   
definition of three fields:

struct [

|  |  |  |  |
| --- | --- | --- | --- |
| unsigned | is\_keyword | : | 1; |
| unsigned | is\_extern | : | 1; |
| unsigned | is\_static | : | 1; |
| } flags; |  |  |  |

This defines a variable called flags that contains three 1-bit fields. The   
number following the colon represents the field width in bits. The fields are   
declared unsigned to emphasize that they really are unsigned quantities.

Individual fields are referenced as flags . is\_keyword,   
flags. is\_extern, etc., just like other structure members. Fields behave   
like small, unsigned integers, and may participate in arithmetic expressions   
just like other integers. Thus the previous examples may be written more   
naturally as

138 THE C PROGRAMMING LANGUAGE CHAPTER 6

flags.is\_extern = flags.is\_static = 1;

to turn the bits on;

flags.is\_extern = flags.is\_static = 0;

to turn them off; and

if (flags.is\_extern == 0 && flags.is\_static == 0) ..

to test them.

A field may not overlap an int boundary; if the width would cause this   
to happen, the field is aligned at the next int boundary. Fields need not be   
named; unnamed fields (a colon and width only) are used for padding. The   
special width 0 may be used to force alignment at the next int boundary.

There are a number of caveats that apply to fields. Perhaps most   
significant, fields are assigned left to right on some machines and right to   
left on others, reflecting the nature of different hardware. This means that   
although fields are quite useful for maintaining internally-defined data struc­   
tures, the question of which end comes first has to be carefully considered   
when picking apart externally-defined data.

Other restrictions to bear in mind: fields are unsigned; they may be   
stored only in it's (or, equivalently, unsigned's); they are not arrays;   
they do not have addresses, so the & operator cannot be applied to them.

6.8 Unions

A *union* is a variable which may hold (at different times) objects of   
different types and sizes, with the compiler keeping track of size and align­   
ment requirements. Unions provide a way to manipulate different kinds of   
data in a single area of storage, without embedding any machine-dependent   
information in the program.

As an example, again from a compiler symbol table, suppose that con­   
stants may be int's, float's or character pointers. The value of a particu­   
lar constant must be stored in a variable of the proper type, yet it is most   
convenient for table management if the value occupies the same amount of   
storage and is stored in the same place regardless of its type. This is the   
purpose of a union — to provide a single variable which can legitimately   
hold any one of several types. As with fields, the syntax is based on struc­   
tures.

union u\_tag (   
int ival;   
float fval;   
char \*pval;

) uval;

CHAPTER 6 STRUCTURES 139

The variable uval will be large enough to hold the largest of the three   
types, regardless of the machine it is compiled on — the code is indepen­   
dent of hardware characteristics. Any one of these types may be assigned to   
uval and then used in expressions, so long as the usage is consistent: the   
type retrieved must be the type most recently stored. It is the responsibility   
of the programmer to keep track of what type is currently stored in a union;   
the results are machine dependent if something is stored as one type and   
extracted as another.

Syntactically, members of a union are accessed as

*union-name, member*

or

*union-pointer—> member*

just as for structures. If the variable utype is used to keep track of the   
current type stored in uval, then one might see code such as

if (utype == INT)

printf("%d\n", uval.ival);

else if (utype == FLOAT)

printf("%f\n", uval.fval);

else if (utype == STRING)

printf("%s\n", uval.pval);

else

printf("bad type %d in utype\n", utype);

Unions may occur within structures and arrays and vice versa. The   
notation for accessing a member of a union in a structure (or vice versa) is   
identical to that for nested structures. For example, in the structure array   
defined by

struct (

char \*name;

int flags;

int utype;

union (

int ival;

float fval;

char \*pval;

) uval;

) symtab[NSYM];

the variable iva 1 is referred to as

symtab[i].uval.ival

and the first character of the string pval by

140 THE C PROGRAMMING LANGUAGE CHAPTER 6

\*symtab[i].uval.pval

In effect, a union is a structure in which all members have offset zero,   
the structure is big enough to hold the "widest" member, and the align­   
ment is appropriate for all of the types in the union. As with structures, the   
only operations currently permitted on unions are accessing a member and   
taking the address; unions may not be assigned to, passed to functions, or   
returned by functions. Pointers to unions can be used in a manner identical   
to pointers to structures.

The storage allocator in Chapter 8 shows how a union can be used to   
force a variable to be aligned on a particular kind of storage boundary.

6.9 Typedef

C provides a facility called typedef for creating new data type names.   
For example, the declaration

typedef int LENGTH;

makes the name LENGTH a synonym for int. The "type" LENGTH can be   
used in declarations, casts, etc., in exactly the same ways that the type int   
can be:

LENGTH len, maxlen;

LENGTH \*lengths[];

Similarly, the declaration

typedef char \*STRING;

makes STRING a synonym for char \* or character pointer, which may   
then be used in declarations like

STRING p, lineptr[LINES], alloc();

Notice that the type being declared in a typedef appears in the posi­   
tion of a variable name, not right after the word typedef. Syntactically,   
typedef is like the storage classes extern, static, etc. We have also   
used upper case letters to emphasize the names.

As a more complicated example, we could make typedef's for the tree   
nodes shown earlier in this chapter:

typedef struct tnode ( /\* the basic node \*/

char \*word; /\* points to the text \*/

int count; /\* number of occurrences \*/

struct tnode \*left; /\* left child \*/

struct tnode \*right; /\* right child \*/   
) TREENODE, \*TREEPTR;

This creates two new type keywords called TREENODE (a structure) and   
TREEPTR (a pointer to the structure). Then the routine talloc could

CHAPTER 6 STRUCTURES 141

become

TREEPTR talloc()

char \*alloc();

return((TREEPTR) alloc(sizeof(TREENODE)));

It must be emphasized that a typedef declaration does not create a   
new type in any sense; it merely adds a new name for some existing type.   
Nor are there any new semantics: variables declared this way have exactly   
the same properties as variables whose declarations are spelled out explicitly.   
In effect, typedef is like #define, except that since it is interpreted by   
the compiler, it can cope with textual substitutions that are beyond the capa­   
bilities of the C macro preprocessor. For example,

typedef int (\*PFI)();

creates the type PFI, for "pointer to function returning int," which can be   
used in contexts like

PFI strcmp, numcmp, swap;

in the sort program of Chapter 5.

There are two main reasons for using typedef declarations. The first   
is to parameterize a program against portability problems. If typedef's are   
used for data types which may be machine dependent, only the typedef's   
need change when the program is moved. One common situation is to use   
typedef names for various integer quantities, then make an appropriate set   
of choices of short, int and long for each host machine.

The second purpose of typedef's is to provide better documentation   
for a program — a type called TREEPTR may be easier to understand than   
one declared only as a pointer to a complicated structure.

Finally, there is always the possibility that in the future the compiler or   
some other program such as *lint* may make use of the information contained   
in typedef declarations to perform some extra checking of a program.