

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.)

While we talk about "data structures", and how to use them in every language, this section is about understanding how software developers carefully control the low-level "shape" of their data items to solve their problems.

When you first learn about the C `struct` keyword, you might think that it is equivalent to a Python `dict` - a dynamic key-value store like a PHP array, Java map, or JavaScript object - but nothing is further from the truth. These other languages provide us with easy-to use data structures where all of the challenging problems are solved.

This chapter tells the creators of Python, PHP, Java, and JavaScript *how* to solve the complex problems and build convenient and flexible data structures which we use in all of those object-oriented languages.

One way to look at the code in this chapter is to think of it as a lesson on how one might build Python's `list` and `dict` data structures. If the code in the chapter takes you a little while to figure out - mentally make a note of thanks for all the hard work that the modern languages invest to make their higher-level data structures so flexible and easy to use.

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 programs, 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.

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 { ... } 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 = { 14, 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 || 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

```
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 forthcoming 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 external or static structures can.

This prediction was indeed accurate. Modern C compilers support the copying of a structure with a single assignment statement. Given that a C structure is just a small fixed length block of memory - it is easy to generate machine code to copy it. A key bit to remember is that when a C structure is copied - it is done as a ["shallow copy"](#). A shallow copy, copies the *values* of the variables and pointers in the structure but does not make copies of any of the data which the pointers point to. A structure contains other structures (i.e. not pointers to structures), then those structures are shallow copied as well.

Let us investigate some of these points by rewriting the date conversion functions of the last chapter to use structures. Since the rules prohibit passing 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 variables.

```
struct date {
    int day;
    int month;
    int year;
    int yearday;
    char mon_name[4];
};

static int day_tab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

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 || pd->year % 400 == 0;
    for (i = 1; i < pd->month; i++)
        day += day_tab[leap][i];
    return (day);
}
```

c_122_01

Copy

Edit

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 provided 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.

```
struct date {
```

```

    int day;
    int month;
    int year;
    int yearday;
    char mon_name[4];
};

static int day_tab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

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 || 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;
}

```

c_123_01

Copy

Edit

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++` increments `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 variables. 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 organization is possible. Each keyword entry is really a pair:

```
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[] = {
    "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 },
...
```

but the inner braces are not necessary when the initializers are simple variables 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 `[]` is left empty.

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.)

```
#include <stdio.h>
#define MAXWORD 20
#define LETTER 'a'
```

c_125_01

Copy

Edit

```

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.

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 terminate 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 structure. 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` statement to set the value of `NKEYS`:

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

Now for the function `getword`. We have actually written a more general `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.

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

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);
}
```

c_127_01

Copy

Edit

`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*.

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

type(c) /* return type of ASCII character */
int c;
{
    if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
        return (LETTER);

    else if (c >= '0' && c <= '9')
        return (DIGIT);
    else
        return(c);
}
```

c_127_02

Copy

Edit

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.

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

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

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

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.

A classic early assignment in any programming language is a "word frequency" program. Here is a Python program from my [Python for Everybody](#) course to count words from an input stream:

```
handle = open('romeo.txt', 'r')
words = handle.read().split()
counts = dict()
for word in words:
    counts[word] = counts.get(word,0) + 1
print(counts)
```

This section implements a less general word counting program in C. The code depends on several functions from earlier in the book, and the code below is pretty complex, where the programmer only has access to a low-level language without powerful and easy-to-use data types like `list` or `dict`.

It is likely that, Guido van Rossum, read this book, took a look at this code and designed the `dict` data structure in Python so that the rest of us could write a data parsing and word frequency program in the above six lines of code without worrying about dynamic memory allocation, pointer management, string length and a myriad of other details that must be solved when solving the problem in C

Since Python is open source, you can actually look at the C code that implements the `dict` object in a file called [dictobject.c](#). It is almost 6000 lines of code and includes 11 other files of utility code. Thankfully we only have to write one line in Python to use it:

```
counts = dict()
```

We will leave the complex bits to the C programmers that build and maintain Python.

This section is not showing us *how* to use the Python `dict` object - rather it is showing how one would *build* a dict-like structure using C.

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);
}

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);
}
```

c_129_01

Copy

Edit

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 middle 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 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 complicated 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

Before we start this section, a slightly longer aside from your narrator. Up to now, I have resisted the temptation to augment the book with my own bits of code. But we have reached the single point in the book where I feel that there is too big of a conceptual leap between two sections so I am going to add some of my own narrative between sections 6.4 and 6.5.

The rest of this chapter talks very nicely about [binary trees](#) and [hash tables](#) - both essential low level data structures in Computer Science and both excellent ways to understand pointers and how C can be used to build data structures like the Python `dict`. However the authors skipped separately describing the structure of a dynamically constructed [linked list](#) which is the first and foundational collection data structure in Computer Science and should be understood before moving to tree and hash map structures.

Linked lists form the foundation of the Python `list` object, Java Array object, PHP numeric key arrays, and JavaScript arrays. The linked list can be dynamically extended and items can be added in the middle efficiently as well as being pushed or popped on or off the front or back of the list. Linked lists also are used to implement queues as well as other aspects of operating systems.

I will attempt to mimic the authors writing style in this new section of the book. I will write the sample code using a more modern dialect of C so it is easier to run on a modern compiler.

6.5.1 Linked Lists (Bonus content)

Suppose we want to read a file and print the file in reverse order. We don't know how many lines will be in the file before we read the file so we can't simply use a simple array of pointers to strings in character arrays (i.e. lines). In a sense we need a dynamic array that grows as we

encounter new lines. When we reach the end of file we just loop through our stored lines from the end to the beginning and print them out in reverse order.

One solution is to make a data structure called a [doubly linked list](#) of character strings. In addition to each line of data, we will store a pointer to the previous line and the next line as well as a pointer to the first item we add to the in the list which we will call the "head" of the list and the most recent item we added to the list which we will call the "tail" of the list.

We will see a single linked list as one part of a hash map data structure below. A single linked list can only be traversed in a forward direction. A doubly linked list can be traversed either forwards or backwards.

Given that our linked list of strings will keep expanding as we get new lines, we avoid hard coding array sizes like

```
#define MAXLEN 1000
```

in the previous chapter when we built a program to sort a file.

Going back to the description of a line in our doubly linked list, it is clearly a structure with four components:

```
struct lnode { /* A line of text */
    char *text; /* points to the text */
    struct lnode *prev; /* The previous line */
    struct lnode *next; /* The next line */
}
```

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

```
struct lnode *prev;
```

declares `prev` to be a *pointer* to an `lnode`, not an `lnode` itself.

We will write the code in a modern C dialect using modern memory allocation and I/O routines provided by the standard C libraries.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXLINE 1000

struct lnode { /* A line of text */
    char *text; /* points to the text */
    struct lnode *prev; /* The previous line */
    struct lnode *next; /* The next line */
};

int main() /* print lines in reverse */
{
    struct lnode *head = NULL;
    struct lnode *tail = NULL;
    struct lnode *current;
    char line[MAXLINE];

    while(fgets(line, MAXLINE, stdin) != NULL) {
```

c_130_01

Copy

Edit

```
char *save = (char *) malloc(strlen(line)+1);
strcpy(save, line);

struct lnode *new = (struct lnode *) malloc(sizeof(struct lnode));
new->text = save;
new->next = NULL;
new->prev = tail;

if ( head == NULL ) head = new;

if ( tail != NULL ) tail->next = new;
tail = new;
}

for (current = tail; current != NULL; current = current->prev ) {
    printf("%s", current->text);
}
}
```

Interestingly if we wanted to print the list in forward order (or if we had a singly linked list), our loop would be as follows:

```
for (current = head; current != NULL; current = current->next ) {
    printf("%s", current->text);
}
```

In general we use the variable names `head`, `tail`, `current` as well as `next` and `prev` or similar names when writing code that builds or uses a linked list so other programmers will quickly understand what we are talking about. After a while, reading a `for` loop to traverse a linked list becomes as natural as reading a `for` loop that progresses through a sequence of numbers.

6.5.2 Binary Trees

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 arbitrary 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 continues 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 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`.

```
#include <stdio.h>
#define MAXWORD 20
#define LETTER 'a'

main() /* word frequency count */
{
    struct tnode *root, *tree();
    char word [MAXWORD];
    int t;

    root = NULL;
    while ((t = get_word(word, MAXWORD)) != EOF)
        if (t == LETTER)
            root = tree(root, word);
    treeprint(root);
}
```

c_131_01

Copy

Edit

`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 incremented), or a null pointer is encountered, indicating that a node must be 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.

```
#include <string.h>
```

c_132_01

Copy

Edit

```

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 */
};

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 recursion, draw yourself a tree and print it with `treeprint`; it's one of the cleanest recursive routines you can find.

```

#include <stdio.h>

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 */
};

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);
    }
}

```

c_133_01

Copy

Edit

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 certain 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.

By now, you know that when the authors mention the PDP-11 they are sharing aspects of the challenge of making C work on previous generation computers with short memory words and small amounts of memory at the same time as making it work on the incoming generation of computers with larger words and more memory.

The research, thought, and care that went into making sure that C code was portable across multiple generations of computer hardware is on display in the previous paragraph.

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 as

```
struct tnode *talloc()
{
    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.

The concerns the authors mention in this section are also nicely resolved in modern C compilers. In the ANSI standard version of C, they introduced the notion of the [void type](#). The `void` type indicates the "lack of" a type - much like `NULL` is used to indicate "not a valid pointer". In 1978 because the `char` type was generally the most native type on any system, it was used when a generic pointer needed to be returned from a memory allocation function. In modern C we use pointers to `void` and then cast the returned pointer to be a pointer to whatever `struct` or other data we just allocated.

If we were writing the `alloc()` routine in this book using modern C, it would return a pointer to a `void`

```
char *alloc(n) /* 1978 version */
int n;
{ ... }

void *alloc(n) /* Modern version */
int n;
{ ... }
```

It is a testament to the foresight of the authors all of the pointer casting code in this book works the same *regardless* of whether memory allocation functions return `char` or `void` pointers to the allocated data.

Exercise 6-4. Write a program which reads a C program and prints in alphabetical 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).

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.

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.

6.6 Table Lookup

In this section, we finish our quick tour of the implementations of the three core data structures in Computer Science: (1) the linked list, (2) the tree, and (3) the hash map (this section). A singly linked list is also part of a hash map implementation so you can compare it to the doubly linked list code introduced earlier in the bonus section 6.5.1.

This section is worth understanding well because not only is it an excellent review of pointers and structures, but also because one of the most common questions on a face-to-face programming interview is, "draw a hash map on the whiteboard and explain how it works". It is an easy question if you study this section of the book and almost impossible if

you have not.

In some way this section is the most intricate data structure that is described in this book - it is why it is so popular on coding interviews. Chapters 7 and 8 talk about practical things like input/output and the UNIX operating system. Elegant data structures and their use are core concepts in Computer Science - understanding them highlights the difference between a good programmer and a Computer Scientist. In a sense, understanding how a hash map works is the "secret handshake" of Computer Science. And it is the secret handshake because of this book and this section of this book written back in 1978 and used in a course that the person interviewing you may have took when they were in college. Hash maps were difficult for them to understand back then - so if you understand the concept you must be solid.

So I hope you pay close attention to this section - and remember the handshake.

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 replacement 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 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.)

```
#define HASHSIZE 100

hash(s) /* form hash value for string s */
char *s;
{
    int hashval;

    for (hashval = 0; *s != '\0'; )
        hashval += *s++;
    return(hashval % HASHSIZE);
}
```

c_135_01

Copy

Edit

Ah hashing functions.

Hashing functions are one of the foundational notions in Computer Science. Hashing functions are used for everything from high-performance in-memory structures, organizing data in databases, digital signing, network packet checksums, security algorithms and much more.

The above text is a great example of a really simple hashing function. You should understand this simple presentation well so that when you encounter a more complex implementation or use of hashing, you can fall back on this text to understand that at its core hashing is a very simple concept.

So much of this chapter is a succinct example of some of the most powerful concepts in Computer Science. Please don't look at eight lines of code above and think "I got that" and jump to the next bit. This chapter is showing you the way of the master programmer. Be patient, slow down, and enjoy your time here.

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 beginning 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`.

```
#include <stdlib.h>
#include <string.h>

struct nlist { /* basic table entry */
    char *name;
    char *def;
    struct nlist *next; /* next entry in chain */
};

#define HASHSIZE 100
static struct nlist *hashtab[HASHSIZE]; /* pointer table */

struct nlist *lookup(s) /* look for s in hashtab */
char *s;
{
    struct nlist *np;
```

c_136_01

Copy

Edit

```

    for (np = hashtab[hash(s)]; np != NULL; np = np->next)
        if (strcmp(s, np->name) == 0)
            return(np); /* found it */
    return(NULL); /* not found */
}

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[hashval];
        hashtab[hashval] = np;
    } else /* already there */
        free(np->def); /* free previous definition */
    if ((np->def = strsave(def)) == NULL)
        return(NULL);
    return(np);
}

```

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

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

`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](#).

One reason the authors make vague forward-looking statements whenever they talk about dynamic memory is that large-scale memory management in a programming language is still a subject of active research 40 years later. Back in 1978 it was absolutely not a settled topic.

You can see this when the authors build a simple non-production memory allocation scheme with their own `alloc()` and `free()` routines backed by a fixed length static extern array of characters. Dynamic allocation is essential to writing competent C programs, but it is likely that production grade dynamic memory support was still somewhat non-portable when the book was written - so they used simple self-contained implementations in this book.

Modern dynamic memory support is through the `malloc()`, `calloc()` and `free()` functions in the standard library. These functions request dynamic memory blocks from the operating system and manage those areas on behalf of your C code. On UNIX and UNIX-like systems the memory allocation layer asks the underlying operating system for blocks of

memory through the [sbrk\(\)](#) interface.

Even with virtual memory, programmers must carefully manage their use of dynamically allocated memory because memory is never unlimited.

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

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

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.

Now we are going to go from low-level programming to even-lower-level programming. The UNIX operating system is written in C and UNIX needs to have an implementation of the Internet protocols so it can be connected to the Internet. One of the most important Internet protocols is the "Transmission Control Protocol" (TCP).

In order to implement TCP, you need to send very precisely formatted data. This data is very tightly packed in order to save precious network bandwidth. The exact format of a TCP header is described in the [TCP Wikipedia page](#). If you look at the header, you will find that at bits 96-99 in the header, TCP expects a four bit integer that defines the "data offset".

Exactly what this data means is less relevant unless you are writing the actual TCP implementation - but it does demonstrate that we need control of our data layout on a bit by bit basis.

This section covers how we can use `struct` to build up a TCP header in C which can be parsed and set without using masking and shifting operators with hard-coded numbers.

The section below is simpler than constructing a valid TCP header using a carefully packed `struct`, but it lays the groundwork for these more complicated situations.

Imagine a fragment of a compiler that manipulates a symbol table. Each identifier in a program has certain information associated with it, for example, 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 |= EXTERNAL | STATIC;
```

turns on the EXTERNAL and STATIC bits in `flags`, while

```
flags &= ~(EXTERNAL | STATIC);
```

turns them off, and

```
if ((flags & (EXTERNAL | 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

```
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 structures, 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 `int`'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 alignment 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 constants may be `int`'s, `float`'s or character pointers. The value of a particular 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 structures.

```
union u_tag {  
    int ival;  
    float fval;  
    char *pval;  
} uval;
```

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 independent 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 `ival` is referred to as

```
symtab[i].uval.ival
```

and the first character of the string `pval` by

```
*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 alignment 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 above limitations on unions are no longer accurate. Like structures, modern C compilers *can* assign the contents of a union to another union variable. You can also pass unions into functions by value and receive a union as the return type of a function.

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 position 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, *TREETPTR;
```

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

```
TREETPTR talloc()  
{  
    char *alloc();  
  
    return((TREETPTR) 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 capabilities 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

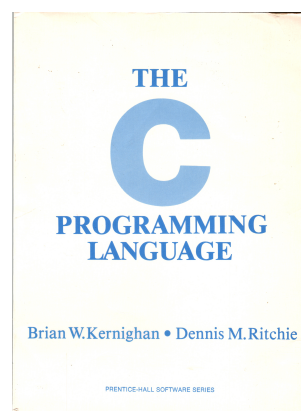
program against portability problems. If `typedef`'s are used for data types which may be machine dependent, only the `typedef`'s need change when the program is moved. One common situation is to use `typedef` names for various integer quantities, then make an appropriate set of choices of `short`, `int` 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.

This work (www.cc4e.com) is based on the 1978 "[The C Programming Language](#)" book written by Brian W. Kernighan and Dennis M. Ritchie. Their book is copyright all rights reserved by AT&T but is being used in this work under "fair use" because of the book's historical and scholarly significance, its lack of availability, and lack of an accessible version of the book.

The book is augmented in places to help understand its rightful place in a historical context amidst the major changes of the 1970's and 1980's as Computer Science evolved from a hardware-first / vendor-centered approach to a software-centered approach where portable operating systems and applications written in C could run on any hardware.



This is not the ideal book to learn C programming because the 1978 edition does not reflect the modern C language. Using an obsolete book gives us an opportunity to take students back in time and understand how the C language was evolving as it laid the ground work for a future with portable applications.

If you want to learn modern C programming from a book that reflects the modern C language, I suggest you use the [C Programming Language, 2nd Edition](#), also written by Brian W. Kernighan and Dennis M. Ritchie, initially published in 1988 and based on the [ANSI C](#) (C89) version of the language.

The source code to this book is at <https://github.com/csev/cc4e>. You are welcome to help in the creation and editing of the book.