

```

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

int getword(char *, int);
struct key *binsearch(char *, struct key *, int);

/* count C keywords; pointer version */
main()
{
    char word[MAXWORD];
    struct key *p;

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((p=binsearch(word, keytab, NKEYS)) != NULL)
                p->count++;
    for (p = keytab; p < keytab + NKEYS; p++)
        if (p->count > 0)
            printf("%4d %s\n", p->count, p->word);
    return 0;
}

/* binsearch: find word in tab[0]...tab[n-1] */
struct key *binsearch(char *word, struct key *tab, int n)
{
    int cond;
    struct key *low = &tab[0];
    struct key *high = &tab[n];
    struct key *mid;

    while (low < high) {
        mid = low + (high-low) / 2;
        if ((cond = strcmp(word, mid->word)) < 0)
            high = mid;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return NULL;
}

```

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

Second, the elements of `keytab` are now accessed by pointers. This

requires significant changes in `binsearch`.

The initializers for `low` and `high` are now pointers to the beginning and just past the end of the table.

The computation of the middle element can no longer be simply

```
mid = (low+high) / 2    /* WRONG */
```

because the addition of two pointers is illegal. Subtraction is legal, however, so `high-low` is the number of elements, and thus

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

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

The most important change is to adjust the algorithm to make sure that it does not generate an illegal pointer or attempt to access an element outside the array. The problem is that `&tab[-1]` and `&tab[n]` are both outside the limits of the array `tab`. The former is strictly illegal, and it is illegal to dereference the latter. The language definition does guarantee, however, that pointer arithmetic that involves the first element beyond the end of an array (that is, `&tab[n]`) will work correctly.

In `main` we wrote

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

If `p` is a pointer to a structure, arithmetic on `p` takes into account the size of the structure, so `p++` increments `p` by the correct amount to get the next element of the array of structures, and the test stops the loop at the right time.

Don't assume, however, 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 unnamed "holes" in a structure. Thus, for instance, if a `char` is one byte and an `int` four bytes, the structure

```
struct {
    char c;
    int i;
};
```

might well require eight bytes, not five. The `sizeof` operator returns the proper value.

Finally, an aside on program format: when a function returns a complicated type like a structure pointer, as in

```
struct key *binsearch(char *word, struct key *tab, int 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 *
binsearch(char *word, struct key *tab, int n)
```

This is 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 too long. (More precisely, its running time is likely to grow quadratically with the number of input words.) How can we organize the data to cope efficiently with a list of arbitrary words?

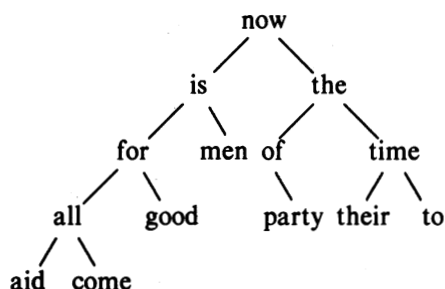
One solution is to keep the set of words seen so far sorted at all times, by placing each word into its proper position in the order as it arrives. This shouldn't be done by shifting words in a linear array, though—that also takes too long. Instead we will use a data structure called a *binary tree*.

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 that are lexicographically less than the word at the node, and the right subtree contains only words that are greater. This is the tree for the sentence “now is the time for all good men to come to the aid of their party”, as built by inserting each word as it is encountered:



To find out whether a new word is already in the tree, start at the root and compare 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, continue searching at the left child, otherwise at the right child. If there is no child in the required direction, the new word is not in the tree, and in fact the empty slot is the proper place to add the new word. This process is 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 conveniently represented as a structure with four components:

```

struct tnode {          /* the tree 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 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 `tnode`, not a `tnode` itself.

Occasionally, one needs a variation of self-referential structures: two structures that refer to each other. The way to handle this is:

```

struct t {
    ...
    struct s *p;    /* p points to an s */
};
struct s {
    ...
    struct t *q;    /* q points to a t */
};

```

The code for the whole program is surprisingly small, given a handful of supporting routines like `getword` that we have already written. The main routine reads words with `getword` and installs them in the tree with `addtree`.

```

#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define MAXWORD 100
struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);

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

    root = NULL;
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            root = addtree(root, word);
    treeprint(root);
    return 0;
}

```

The function `addtree` is recursive. 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 `addtree`. 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, `addtree` returns a pointer to it, which is installed in the parent node.

```

struct tnode *talloc(void);
char *strdup(char *);

/* addtree: add a node with w, at or below p */
struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;

    if (p == NULL) {      /* a new word has arrived */
        p = talloc();     /* make a new node */
        p->word = strdup(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)     /* less than into left subtree */
        p->left = addtree(p->left, w);
    else                  /* greater than into right subtree */
        p->right = addtree(p->right, w);
    return p;
}

```

Storage for the new node is fetched by a routine `talloc`, which returns a pointer to a free space suitable for holding a tree node, and the new word is copied to a hidden place by `strdup`. (We will discuss these routines in a moment.) The count is initialized, and the two children are made null. This part of the code is executed only at the leaves of the tree, when a new node is being added. We have (unwisely) omitted error checking on the values returned by `strdup` and `talloc`.

`treeprint` prints the tree in sorted 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 how recursion works, simulate `treeprint` as it operates on the tree shown above.

```

/* treeprint: in-order print of tree p */
void treeprint(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 much. 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 that 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 chars and pointers to struct tnodes, 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 at even addresses)? Second, what declarations can cope with the fact that an allocator must necessarily return different kinds of pointers?

Alignment requirements can generally be satisfied easily, at the cost of some wasted space, by ensuring that the allocator always returns a pointer that meets *all* alignment restrictions. The `alloc` of Chapter 5 does not guarantee any particular alignment, so we will use the standard library function `malloc`, which does. In Chapter 8 we will show one way to implement `malloc`.

The question of the type declaration for a function like `malloc` is a vexing one for any language that takes its type-checking seriously. In C, the proper method is to declare that `malloc` returns a pointer to `void`, then explicitly coerce the pointer into the desired type with a cast. `malloc` and related routines are declared in the standard header `<stdlib.h>`. Thus `talloc` can be written as

```

#include <stdlib.h>

/* talloc: make a tnode */
struct tnode *talloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}

```

`strdup` merely copies the string given by its argument into a safe place, obtained by a call on `malloc`:

```

char *strdup(char *s)    /* make a duplicate of s */
{
    char *p;

    p = (char *) malloc(strlen(s)+1); /* +1 for '\0' */
    if (p != NULL)
        strcpy(p, s);
    return p;
}

```

`malloc` returns `NULL` if no space is available; `strdup` passes that value on, leaving error-handling to its caller.

Storage obtained by calling `malloc` may be freed for re-use by calling `free`; see Chapters 7 and 8.

Exercise 6-2. Write a program that reads a C program and prints in alphabetical order each group of variable names that are identical in the first 6 characters, but different somewhere thereafter. Don't count words within strings and comments. Make 6 a parameter that can be set from the command line. □

Exercise 6-3. Write a cross-referencer that prints a list of all words in a document, and, for each word, a list of the line numbers on which it occurs. Remove noise words like "the," "and," and so on. □

Exercise 6-4. Write a program that 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 will write the innards of a table-lookup package, to illustrate 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 `#define` statement. When a line like

```
#define IN 1
```

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

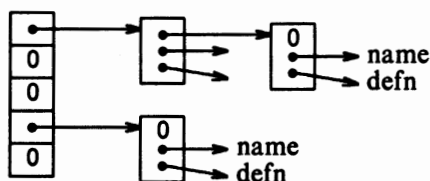
```
state = IN;
```

it must be replaced by `1`.

There are two 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 is a hash search—the incoming name is converted into a small

non-negative integer, which is then used to index into an array of pointers. An array element points to the beginning of a linked list of blocks describing names that have that hash value. It is NULL if no names have hashed to that value.



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

```

struct nlist {          /* table entry: */
    struct nlist *next; /* next entry in chain */
    char *name;         /* defined name */
    char *defn;         /* replacement text */
};
  
```

The pointer array is just

```

#define HASHSIZE 101

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

The hashing function, which is used by both `lookup` and `install`, adds each character value in the string to a scrambled combination of the previous ones and returns the remainder modulo the array size. This is not the best possible hash function, but it is short and effective.

```

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

    for (hashval = 0; *s != '\0'; s++)
        hashval = *s + 31 * hashval;
    return hashval % HASHSIZE;
}
  
```

Unsigned arithmetic ensures that the hash value is non-negative.

The hashing process produces a starting index in the array `hashtab`; if the string is to be found anywhere, it will be in the list 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.


```

/* lookup: look for s in hashtable */
struct nlist *lookup(char *s)
{
    struct nlist *np;

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

```

The for loop in lookup is the standard idiom for walking along a linked list:

```

for (ptr = head; ptr != NULL; ptr = ptr->next)
    ...

```

install uses lookup to determine whether the name being installed is already present; if so, the new definition will supersede the old one. Otherwise, a new entry is created. install returns NULL if for any reason there is no room for a new entry.

```

struct nlist *lookup(char *);
char *strdup(char *);

/* install: put (name, defn) in hashtable */
struct nlist *install(char *name, char *defn)
{
    struct nlist *np;
    unsigned hashval;

    if ((np = lookup(name)) == NULL) { /* not found */
        np = (struct nlist *) malloc(sizeof(*np));
        if (np == NULL || (np->name = strdup(name)) == NULL)
            return NULL;
        hashval = hash(name);
        np->next = hashtable[hashval];
        hashtable[hashval] = np;
    } else /* already there */
        free((void *) np->defn); /* free previous defn */
    if ((np->defn = strdup(defn)) == NULL)
        return NULL;
    return np;
}

```

Exercise 6-5. Write a function undef that will remove a name and definition from the table maintained by lookup and install. □

Exercise 6-6. Implement a simple version of the #define processor (i.e., no arguments) suitable for use with C programs, based on the routines of this section. You may also find getch and ungetch helpful. □

6.7 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 and casts:

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

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 used capitalized names for `typedefs`, to make them stand out.

As a more complicated example, we could make `typedefs` for the tree nodes shown earlier in this chapter:

```
typedef struct tnode *Treenptr;

typedef struct tnode { /* the tree node: */
    char *word;          /* points to the text */
    int count;           /* number of occurrences */
    Treenptr left;       /* left child */
    Treenptr right;      /* right child */
} Treenode;
```

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

```
Treenptr talloc(void)
{
    return (Treenptr) malloc(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 preprocessor. For example,

```
typedef int (*PFI)(char *, char *);
```

creates the type PFI, for “pointer to function (of two char * arguments) returning int,” which can be used in contexts like

```
PFI strcmp, numcmp;
```

in the sort program of Chapter 5.

Besides purely aesthetic issues, there are two main reasons for using typedefs. The first is to parameterize a program against portability problems. If typedefs are used for data types that may be machine-dependent, only the typedefs 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. Types like size_t and ptrdiff_t from the standard library are examples.

The second purpose of typedefs 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.

6.8 Unions

A *union* is a variable that 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. They are analogous to variant records in Pascal.

As an example such as might be found in a compiler symbol table manager, suppose that a constant may be an int, a float, or a character pointer. 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—a single variable that can legitimately hold any one of several types. The syntax is based on structures:

```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```

The variable u will be large enough to hold the largest of the three types; the specific size is implementation-dependent. Any one of these types may be assigned to u 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 programmer's

responsibility to keep track of which type is currently stored in a union; the results are implementation-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 *u*, then one might see code such as

```
if (utype == INT)
    printf("%d\n", u.ival);
else if (utype == FLOAT)
    printf("%f\n", u.fval);
else if (utype == STRING)
    printf("%s\n", u.sval);
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 *sval;
    } u;
} symtab[NSYM];
```

the member *ival* is referred to as

symtab[i].u.ival

and the first character of the string *sval* by either of

```
*symtab[i].u.sval
symtab[i].u.sval[0]
```

In effect, a union is a structure in which all members have offset zero from the base, the structure is big enough to hold the "widest" member, and the alignment is appropriate for all of the types in the union. The same operations are permitted on unions as on structures: assignment to or copying as a unit, taking the address, and accessing a member.

A union may only be initialized with a value of the type of its first member;

thus the union `u` described above can only be initialized with an integer value.

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 Bit-fields

When storage space is at a premium, it may be necessary to pack several objects into a single machine word; one 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.

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
```

or

```
enum { KEYWORD = 01, EXTERNAL = 02, 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 that 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 *bit-field*, or *field* for short, is a set of adjacent bits within a single implementation-defined storage unit that we will call a “word.” The syntax of field definition and access is based on structures. For example, the symbol table `#defines` above could be replaced by the definition of three

fields:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern  : 1;
    unsigned int 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 int` to ensure that they are unsigned quantities.

Individual fields are referenced in the same way as other structure members: `flags.is_keyword`, `flags.is_extern`, etc. Fields behave like small 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.

Almost everything about fields is implementation-dependent. Whether a field may overlap a word boundary is implementation-defined. 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 word boundary.

Fields are assigned left to right on some machines and right to left on others. This means that although fields are 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; programs that depend on such things are not portable. Fields may be declared only as `ints`; for portability, specify `signed` or `unsigned` explicitly. They are not arrays, and they do not have addresses, so the `&` operator cannot be applied to them.

CHAPTER 7: **Input and Output**

Input and output facilities are not part of the C language itself, so we have not emphasized them in our presentation thus far. Nonetheless, programs interact with their environment in much more complicated ways than those we have shown before. In this chapter we will describe the standard library, a set of functions that provide input and output, string handling, storage management, mathematical routines, and a variety of other services for C programs. We will concentrate on input and output.

The ANSI standard defines these library functions precisely, so that they can exist in compatible form on any system where C exists. Programs that confine their system interactions to facilities provided by the standard library can be moved from one system to another without change.

The properties of library functions are specified in more than a dozen headers; we have already seen several of these, including `<stdio.h>`, `<string.h>`, and `<ctype.h>`. We will not present the entire library here, since we are more interested in writing C programs that use it. The library is described in detail in Appendix B.

7.1 Standard Input and Output

As we said in Chapter 1, the library implements a simple model of text input and output. A text stream consists of a sequence of lines; each line ends with a newline character. If the system doesn't operate that way, the library does whatever is necessary to make it appear as if it does. For instance, the library might convert carriage return and linefeed to newline on input and back again on output.

The simplest input mechanism is to read one character at a time from the *standard input*, normally the keyboard, with `getchar`:

```
int getchar(void)
```

`getchar` returns the next input character each time it is called, or EOF when it encounters end of file. The symbolic constant `EOF` is defined in `<stdio.h>`.

The value is typically `-1`, but tests should be written in terms of `EOF` so as to be independent of the specific value.

In many environments, a file may be substituted for the keyboard by using the `<` convention for input redirection: if a program `prog` uses `getchar`, then the command line

```
prog <infile
```

causes `prog` to read characters from `infile` instead. The switching of the input is done in such a way that `prog` itself is oblivious to the change; in particular, the string `"<infile"` is not included in the command-line arguments in `argv`. Input switching is also invisible if the input comes from another program via a pipe mechanism: on some systems, the command line

```
otherprog | prog
```

runs the two programs `otherprog` and `prog`, and pipes the standard output of `otherprog` into the standard input for `prog`.

The function

```
int putchar(int)
```

is used for output: `putchar(c)` puts the character `c` on the *standard output*, which is by default the screen. `putchar` returns the character written, or `EOF` if an error occurs. Again, output can usually be directed to a file with `>filename`: if `prog` uses `putchar`,

```
prog >outfile
```

will write the standard output to `outfile` instead. If pipes are supported,

```
prog | anotherprog
```

puts the standard output of `prog` into the standard input of `anotherprog`.

Output produced by `printf` also finds its way to the standard output. Calls to `putchar` and `printf` may be interleaved—output appears in the order in which the calls were made.

Each source file that refers to an input/output library function must contain the line

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

before the first reference. When the name is bracketed by `<` and `>` a search is made for the header in a standard set of places (for example, on UNIX systems, typically in the directory `/usr/include`).

Many programs read only one input stream and write only one output stream; for such programs, input and output with `getchar`, `putchar`, and `printf` may be entirely adequate, and is certainly enough to get started. This is particularly true if redirection is used to connect the output of one program to the input of the next. For example, consider the program `lower`, which converts its input to lower case:


```
#include <stdio.h>
#include <ctype.h>

main() /* lower: convert input to lower case */
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

The function `tolower` is defined in `<ctype.h>`; it converts an upper case letter to lower case, and returns other characters untouched. As we mentioned earlier, “functions” like `getchar` and `putchar` in `<stdio.h>` and `tolower` in `<ctype.h>` are often macros, thus avoiding the overhead of a function call per character. We will show how this is done in Section 8.5. Regardless of how the `<ctype.h>` functions are implemented on a given machine, programs that use them are shielded from knowledge of the character set.

Exercise 7-1. Write a program that converts upper case to lower or lower case to upper, depending on the name it is invoked with, as found in `argv[0]`. □

7.2 Formatted Output—Printf

The output function `printf` translates internal values to characters. We have used `printf` informally in previous chapters. The description here covers most typical uses but is not complete; for the full story, see Appendix B.

```
int printf(char *format, arg1, arg2, ...)
```

`printf` converts, formats, and prints its arguments on the standard output under control of the format. It returns the number of characters printed.

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to `printf`. Each conversion specification begins with a % and ends with a conversion character. Between the % and the conversion character there may be, in order:

- A minus sign, which specifies left adjustment of the converted argument.
- A number that specifies the minimum field width. The converted argument will be printed in a field at least this wide. If necessary it will be padded on the left (or right, if left adjustment is called for) to make up the field width.
- A period, which separates the field width from the precision.
- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits after the decimal point of a floating-point value, or the minimum number of digits for an integer.

- An `h` if the integer is to be printed as a short, or `l` (letter ell) if as a long.

Conversion characters are shown in Table 7-1. If the character after the `%` is not a conversion specification, the behavior is undefined.

TABLE 7-1. BASIC PRINTF CONVERSIONS

CHARACTER	ARGUMENT TYPE; PRINTED AS
<code>d, i</code>	int; decimal number.
<code>o</code>	int; unsigned octal number (without a leading zero).
<code>x, X</code>	int; unsigned hexadecimal number (without a leading <code>0x</code> or <code>0X</code>), using <code>abcdef</code> or <code>ABCDEF</code> for 10, ..., 15.
<code>u</code>	int; unsigned decimal number.
<code>c</code>	int; single character.
<code>s</code>	char *: print characters from the string until a <code>'\0'</code> or the number of characters given by the precision.
<code>f</code>	double; <code>[-]m.dddddd</code> , where the number of <code>d</code> 's is given by the precision (default 6).
<code>e, E</code>	double; <code>[-]m.dddddd e±xx</code> or <code>[-]m.dddddd E±xx</code> , where the number of <code>d</code> 's is given by the precision (default 6).
<code>g, G</code>	double; use <code>%e</code> or <code>%E</code> if the exponent is less than <code>-4</code> or greater than or equal to the precision; otherwise use <code>%f</code> . Trailing zeros and a trailing decimal point are not printed.
<code>p</code>	void *: pointer (implementation-dependent representation).
<code>%</code>	no argument is converted; print a <code>%</code> .

A width or precision may be specified as `*`, in which case the value is computed by converting the next argument (which must be an int). For example, to print at most `max` characters from a string `s`,

```
printf("%.s", max, s);
```

Most of the format conversions have been illustrated in earlier chapters. One exception is precision as it relates to strings. The following table shows the effect of a variety of specifications in printing "hello, world" (12 characters). We have put colons around each field so you can see its extent.

```

:s:           :hello, world:
%10s:         :hello, world:
%.10s:         :hello, wor:
%-10s:         :hello, world:
%.15s:         :hello, world:
%-15s:         :hello, world :
%15.10s:       :  hello, wor:
%-15.10s:      :hello, wor  :
```

A warning: `printf` uses its first argument to decide how many arguments

follow and what their types are. It will get confused, and you will get wrong answers, if there are not enough arguments or if they are the wrong type. You should also be aware of the difference between these two calls:

```
printf(s);          /* FAILS if s contains % */  
printf("%s", s);    /* SAFE */
```

The function `sprintf` does the same conversions as `printf` does, but stores the output in a string:

```
int sprintf(char *string, char *format, arg1, arg2, ...)
```

`sprintf` formats the arguments in `arg1`, `arg2`, etc., according to `format` as before, but places the result in `string` instead of on the standard output; `string` must be big enough to receive the result.

Exercise 7-2. Write a program that will print arbitrary input in a sensible way. As a minimum, it should print non-graphic characters in octal or hexadecimal according to local custom, and break long text lines. □

7.3 Variable-length Argument Lists

This section contains an implementation of a minimal version of `printf`, to show how to write a function that processes a variable-length argument list in a portable way. Since we are mainly interested in the argument processing, `minprintf` will process the format string and arguments but will call the real `printf` to do the format conversions.

The proper declaration for `printf` is

```
int printf(char *fmt, ...)
```

where the declaration `...` means that the number and types of these arguments may vary. The declaration `...` can only appear at the end of an argument list. Our `minprintf` is declared as

```
void minprintf(char *fmt, ...)
```

since we will not return the character count that `printf` does.

The tricky bit is how `minprintf` walks along the argument list when the list doesn't even have a name. The standard header `<stdarg.h>` contains a set of macro definitions that define how to step through an argument list. The implementation of this header will vary from machine to machine, but the interface it presents is uniform.

The type `va_list` is used to declare a variable that will refer to each argument in turn; in `minprintf`, this variable is called `ap`, for "argument pointer." The macro `va_start` initializes `ap` to point to the first unnamed argument. It must be called once before `ap` is used. There must be at least one named argument; the final named argument is used by `va_start` to get started.

Each call of `va_arg` returns one argument and steps ap to the next; `va_arg` uses a type name to determine what type to return and how big a step to take. Finally, `va_end` does whatever cleanup is necessary. It must be called before the function returns.

These properties form the basis of our simplified `printf`:

```
#include <stdarg.h>

/* minprintf: minimal printf with variable argument list */
void minprintf(char *fmt, ...)
{
    va_list ap; /* points to each unnamed arg in turn */
    char *p, *sval;
    int ival;
    double dval;

    va_start(ap, fmt); /* make ap point to 1st unnamed arg */
    for (p = fmt; *p; p++) {
        if (*p != '%') {
            putchar(*p);
            continue;
        }
        switch (++p) {
            case 'd':
                ival = va_arg(ap, int);
                printf("%d", ival);
                break;
            case 'f':
                dval = va_arg(ap, double);
                printf("%f", dval);
                break;
            case 's':
                for (sval = va_arg(ap, char *); *sval; sval++)
                    putchar(*sval);
                break;
            default:
                putchar(*p);
                break;
        }
    }
    va_end(ap); /* clean up when done */
}
```

Exercise 7-3. Revise `minprintf` to handle more of the other facilities of `printf`. □

7.4 Formatted Input—Scanf

The function `scanf` is the input analog of `printf`, providing many of the same conversion facilities in the opposite direction.

```
int scanf(char *format, ...)
```

`scanf` reads characters from the standard input, interprets them according to the specification in `format`, and stores the results through the remaining arguments. The format argument is described below; the other arguments, *each of which must be a pointer*, indicate where the corresponding converted input should be stored. As with `printf`, this section is a summary of the most useful features, not an exhaustive list.

`scanf` stops when it exhausts its format string, or when some input fails to match the control specification. It returns as its value the number of successfully matched and assigned input items. This can be used to decide how many items were found. On end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match the first specification in the format string. The next call to `scanf` resumes searching immediately after the last character already converted.

There is also a function `sscanf` that reads from a string instead of the standard input:

```
int sscanf(char *string, char *format, arg1, arg2, ...)
```

It scans the `string` according to the format in `format`, and stores the resulting values through `arg1`, `arg2`, etc. These arguments must be pointers.

The format string usually contains conversion specifications, which are used to control conversion of input. The format string may contain:

- Blanks or tabs, which are ignored.
- Ordinary characters (not %), which are expected to match the next non-white space character of the input stream.
- Conversion specifications, consisting of the character %, an optional assignment suppression character *, an optional number specifying a maximum field width, an optional h, l, or L indicating the width of the target, and a conversion character.

A conversion specification directs the conversion of the next input field. Normally the result is placed in the variable pointed to by the corresponding argument. If assignment suppression is indicated by the * character, however, the input field is skipped; no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. This implies that `scanf` will read across line boundaries to find its input, since newlines are white space. (White space characters are blank, tab, newline, carriage return, vertical tab, and formfeed.)

The conversion character indicates the interpretation of the input field. The corresponding argument must be a pointer, as required by the call-by-value

semantics of C. Conversion characters are shown in Table 7-2.

TABLE 7-2. BASIC SCANF CONVERSIONS

CHARACTER	INPUT DATA; ARGUMENT TYPE
d	decimal integer; int *.
i	integer; int *. The integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X).
o	octal integer (with or without leading zero); int *.
u	unsigned decimal integer; unsigned int *.
x	hexadecimal integer (with or without leading 0x or 0X); int *.
c	characters; char *. The next input characters (default 1) are placed at the indicated spot. The normal skip over white space is suppressed; to read the next non-white space character, use %1s.
s	character string (not quoted); char *, pointing to an array of characters large enough for the string and a terminating '\0' that will be added.
e, f, g	floating-point number with optional sign, optional decimal point and optional exponent; float *.
%	literal %; no assignment is made.

The conversion characters d, i, o, u, and x may be preceded by h to indicate that a pointer to short rather than int appears in the argument list, or by l (letter ell) to indicate that a pointer to long appears in the argument list. Similarly, the conversion characters e, f, and g may be preceded by l to indicate that a pointer to double rather than float is in the argument list.

As a first example, the rudimentary calculator of Chapter 4 can be written with scanf to do the input conversion:

```
#include <stdio.h>

main() /* rudimentary calculator */
{
    double sum, v;

    sum = 0;
    while (scanf("%lf", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

Suppose we want to read input lines that contain dates of the form

25 Dec 1988

The scanf statement is

```
int day, year;
char monthname[20];

scanf("%d %s %d", &day, monthname, &year);
```

No & is used with monthname, since an array name is a pointer.

Literal characters can appear in the `scanf` format string; they must match the same characters in the input. So we could read dates of the form mm/dd/yy with this `scanf` statement:

```
int day, month, year;

scanf("%d/%d/%d", &month, &day, &year);
```

`scanf` ignores blanks and tabs in its format string. Furthermore, it skips over white space (blanks, tabs, newlines, etc.) as it looks for input values. To read input whose format is not fixed, it is often best to read a line at a time, then pick it apart with `sscanf`. For example, suppose we want to read lines that might contain a date in either of the forms above. Then we could write

```
while (getline(line, sizeof(line)) > 0) {
    if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
        printf("valid: %s\n", line);    /* 25 Dec 1988 form */
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
        printf("valid: %s\n", line);    /* mm/dd/yy form */
    else
        printf("invalid: %s\n", line); /* invalid form */
}
```

Calls to `scanf` can be mixed with calls to other input functions. The next call to any input function will begin by reading the first character not read by `scanf`.

A final warning: the arguments to `scanf` and `sscanf` *must* be pointers. By far the most common error is writing

```
scanf("%d", n);
```

instead of

```
scanf("%d", &n);
```

This error is not generally detected at compile time.

Exercise 7-4. Write a private version of `scanf` analogous to `minprintf` from the previous section. □

Exercise 7-5. Rewrite the postfix calculator of Chapter 4 to use `scanf` and/or `sscanf` to do the input and number conversion. □

7.5 File Access

The examples so far have all read the standard input and written the standard output, which are automatically defined for a program by the local operating system.

The next step is to write a program that accesses a file that is *not* already connected to the program. One program that illustrates the need for such operations is `cat`, which concatenates a set of named files onto the standard output. `cat` is used for printing files on the screen, and as a general-purpose input collector for programs that do not have the capability of accessing files by name. For example, the command

```
cat x.c y.c
```

prints the contents of the files `x.c` and `y.c` (and nothing else) on the standard output.

The question is how to arrange for the named files to be read—that is, how to connect the external names that a user thinks of to the statements that read the data.

The rules are simple. Before it can be read or written, a file has to be *opened* by the library function `fopen`. `fopen` takes an external name like `x.c` or `y.c`, does some housekeeping and negotiation with the operating system (details of which needn't concern us), and returns a pointer to be used in subsequent reads or writes of the file.

This pointer, called the *file pointer*, points to a structure that contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and whether errors or end of file have occurred. Users don't need to know the details, because the definitions obtained from `<stdio.h>` include a structure declaration called `FILE`. The only declaration needed for a file pointer is exemplified by

```
FILE *fp;  
FILE *fopen(char *name, char *mode);
```

This says that `fp` is a pointer to a `FILE`, and `fopen` returns a pointer to a `FILE`. Notice that `FILE` is a type name, like `int`, not a structure tag; it is defined with a `typedef`. (Details of how `fopen` can be implemented on the UNIX system are given in Section 8.5.)

The call to `fopen` in a program is

```
fp = fopen(name, mode);
```

The first argument of `fopen` is a character string containing the name of the file. The second argument is the *mode*, also a character string, which indicates how one intends to use the file. Allowable modes include read ("`r`"), write ("`w`"), and append ("`a`"). Some systems distinguish between text and binary files; for the latter, a "`b`" must be appended to the mode string.

If a file that does not exist is opened for writing or appending, it is created if possible. Opening an existing file for writing causes the old contents to be discarded, while opening for appending preserves them. Trying to read a file that does not exist is an error, and there may be other causes of error as well, like trying to read a file when you don't have permission. If there is any error, `fopen` will return `NULL`. (The error can be identified more precisely; see the discussion of error-handling functions at the end of Section 1 in Appendix B.)

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which `getc` and `putc` are the simplest. `getc` returns the next character from a file; it needs the file pointer to tell it which file.

```
int getc(FILE *fp)
```

`getc` returns the next character from the stream referred to by `fp`; it returns `EOF` for end of file or error.

`putc` is an output function:

```
int putc(int c, FILE *fp)
```

`putc` writes the character `c` to the file `fp` and returns the character written, or `EOF` if an error occurs. Like `getchar` and `putchar`, `getc` and `putc` may be macros instead of functions.

When a C program is started, the operating system environment is responsible for opening three files and providing file pointers for them. These files are the standard input, the standard output, and the standard error; the corresponding file pointers are called `stdin`, `stdout`, and `stderr`, and are declared in `<stdio.h>`. Normally `stdin` is connected to the keyboard and `stdout` and `stderr` are connected to the screen, but `stdin` and `stdout` may be redirected to files or pipes as described in Section 7.1.

`getchar` and `putchar` can be defined in terms of `getc`, `putc`, `stdin`, and `stdout` as follows:

```
#define getchar()    getc(stdin)
#define putchar(c)   putc((c), stdout)
```

For formatted input or output of files, the functions `fscanf` and `fprintf` may be used. These are identical to `scanf` and `printf`, except that the first argument is a file pointer that specifies the file to be read or written; the format string is the second argument.

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)
```

With these preliminaries out of the way, we are now in a position to write the program `cat` to concatenate files. The design is one that has been found convenient for many programs. If there are command-line arguments, they are interpreted as filenames, and processed in order. If there are no arguments, the standard input is processed.