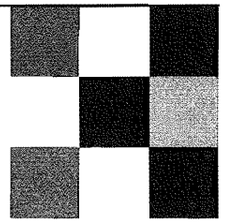
33 If you have a small amount of data to be sorted, a bubble sort or transposition sort works fine. If you have more data, then `qsort()` can be used. Although `qsort()` certainly is a lot faster than a bubble sort, for most implementations it is not as fast as `quicksort()`. The advantage of `qsort()` over `quicksort()` is that it can be implemented with just a few lines of code. The advantage of `quicksort()` over `qsort()` is that it is often faster. By how much? On our system, the answer is that it is a *lot* faster. Write a program to test this. In your program, declare two large arrays. Fill the first array with randomly distributed integers and copy it into the second array. Time how long it takes `qsort()` to sort the first array and how long it takes `quicksort()` to sort the second array. Print these two values and their quotient. Because of system overhead, the size of the quotient will depend on the size of the arrays being sorted. It will also depend on how much effort you put into fine-tuning your quicksort algorithm. In any case, remember what quotients you get (or jot them down in the margin).

34 In the previous exercise, you computed the quotient of running times required to sort a large array, first with `qsort()` and then with `quicksort()`. In this exercise, you are to compute those quotients again, first with your array elements randomly distributed in the interval [0, 100000] and then with array elements randomly distributed in the interval [0, 1]. The results of this exercise can be quite surprising. Try it.



Chapter 9

Structures and Unions

C is an easily extensible language. It can be extended by providing macros that are stored in header files and by providing functions that are stored in libraries. It can also be extended by defining data types that are constructed from the fundamental types. An array type is an example of this; it is a derived type that is used to represent homogeneous data. In contrast, the structure type is used to represent heterogeneous data. A structure has components, called *members*, that are individually named. Because the members of a structure can be of various types, the programmer can create aggregates of data that are suitable for a particular application.

9.1 Structures

The structure mechanism provides a means to aggregate variables of different types. As a simple example, let us define a structure that describes a playing card. The spots on a card that represent its numeric value are called "pips." A playing card such as the three of spades has a pip value, 3, and a suit value, spades. We can declare the structure type

```
struct card {
   int    pips;
   char   suit;
};
```

to capture the information needed to represent a playing card. In this declaration, `struct` is a keyword, `card` is the structure tag name, and the variables `pips` and `suit` are members of the structure. The variable `pips` will take values from 1 to 13, repre-

senting ace to king; the variable `suit` will take values from `'c'`, `'d'`, `'h'`, and `'s'`, representing the suits clubs, diamonds, hearts, and spades, respectively.

This declaration creates the derived data type `struct card`. It is an example of a user-defined type. The declaration can be thought of as a template; it creates the type `struct card`, but no storage is allocated. The tag name, along with the keyword `struct`, can now be used to declare variables of this type.

```
struct card   c1, c2;
```

This declaration allocates storage for the identifiers `c1` and `c2`, which are of type `struct card`. An alternative scheme is to write

```
struct card {
   int    pips;
   char   suit;
} c1, c2;
```

which defines the type `struct card` and declares `c1` and `c2` to be of this type, all at the same time.

To access the members of a structure, we use the member access operator ".". Let us assign to `c1` the values representing the three of spades.

```
c1.pips = 3;
c1.suit = 's';
```

A construct of the form

structure_variable . *member_name*

is used as a variable in the same way that a simple variable or an element of an array is used. If we want `c2` to represent the same playing card as `c1`, then we can write

```
c2 = c1;
```

This causes each member of `c2` to be assigned the value of the corresponding member of `c1`.

Programmers commonly use the `typedef` mechanism when using structure types. An example of this is

```
typedef   struct card   card;
```

Now, if we want more variables to represent playing cards, we can write

```
card   c3, c4, c5;
```


Note that in the type definition the identifier `card` is used twice. In C, the name space for tags is separate from that of other identifiers. Thus, the type definition for `card` is appropriate.

Within a given structure, the member names must be unique. However, members in different structures are allowed to have the same name. This does not create confusion because a member is always accessed through a structure identifier or expression. Consider the following code:

```
struct fruit {
   char   *name;
   int    calories;
};

struct vegetable {
   char   *name;
   int    calories;
};

struct fruit       a;
struct vegetable   b;
```

Having made these declarations, it is clear that we can access `a.calories` and `b.calories` without ambiguity.

Structures can be complicated. They can contain members that are themselves arrays or structures. Also, we can have arrays of structures. Before presenting some examples, let us give the syntax of a structure declaration:

structure_declaration ::= *struct_specifier* *declarator_list* ;
struct_specifier ::= `struct` *tag_name*
                     | `struct` tag_name_{opt} { { *member_declaration* $}_{1+}$ }
tag_name ::= *identifier*
member_declaration ::= *type_specifier* *declarator_list* ;
declarator_list ::= *declarator* { , *declarator* $}_{0+}$

An example is

```
struct card {
   int    pips;
   char   suit;
} deck[52];
```

Here, the identifier `deck` is declared to be an array of `struct card`.

If a tag name is not supplied, then the structure type cannot be used in later declarations. An example is

```
struct {
   int    day, month, year;
   char   day_name[4];          /* Mon, Tue, Wed, etc. */
   char   month_name[4];        /* Jan, Feb, Mar, etc. */
} yesterday, today, tomorrow;
```

This declares `yesterday`, `today`, and `tomorrow` to represent three dates. Because a tag name is not present, more variables of this type cannot be declared later. In contrast, the declaration

```
struct date {
   int    day, month, year;
   char   day_name[4];          /* Mon, Tue, Wed, etc. */
   char   month_name[4];        /* Jan, Feb, Mar, etc. */
};
```

has `date` as a structure tag name, but no variables are declared of this type. We think of this as a template. We can write

```
struct date   yesterday, today, tomorrow;
```

to declare variables of this type.

It is usually good programming practice to associate a tag name with a structure type. It is both convenient for further declarations and for documentation. However, when using `typedef` to name a structure type, the tag name may be unimportant. An example is

```
typedef struct {
   float   re;
   float   im;
} complex;

complex   a, b, c[100];
```

The type `complex` now serves in place of the structure type. The programmer achieves a high degree of modularity and portability by using `typedef` to name such derived types and by storing them in header files.

9.2 Accessing Members of a Structure

In this section we discuss methods for accessing members of a structure. We have already seen the use of the member access operator ".". We will give further examples of its use and introduce the member access operator ->.

Suppose we are writing a program called *class_info*, which generates information about a class of 100 students. We begin by creating a header file.

In file class_info.h

```
#define   CLASS_SIZE   100

struct student {
   char   *last_name;
   int    student_id;
   char   grade;
};
```

This header file can now be used to share information with the modules making up the program. Suppose in another file we write

```
#include "class_info.h"

int main(void)
{
   struct student   tmp, class[CLASS_SIZE];
   .....
```

We can assign values to the members of the structure variable `tmp` by using statements such as

```
tmp.grade = 'A';
tmp.last_name = "Casanova";
tmp.student_id = 910017;
```

Now suppose we want to count the number of failing students in a given class. To do this, we write a function named `fail()` that counts the number of F grades in the array `class[]`. The `grade` member of each element in the array of structures must be accessed.

```
/* Count the failing grades. */

#include "class_info.h"

int fail(struct student class[])
{
   int   i, cnt = 0;

   for (i = 0; i < CLASS_SIZE; ++i)
      cnt += class[i].grade == 'F';
   return cnt;
}
```

Dissection of the `fail()` Function

■
```
int fail(struct student class[])
{
   int   i, cnt = 0;
```

The parameter `class` is of type pointer to `struct student`. An equivalent declaration for this parameter would be

```
struct student *class
```

We can think of `class` as a one-dimensional array of structures. Parameters of any type, including structure types, can be used in headers to function definitions.

■
```
for (i = 0; i < CLASS_SIZE; ++i)
```

We are assuming that when this function is called, an array of type `struct student` of size `CLASS_SIZE` will be passed as an argument.

■
```
cnt += class[i].grade == 'F';
```

An expression such as this demonstrates how concise C can be. C is operator rich. To be fluent in its use, the programmer must be careful about precedence and associativity. This statement is equivalent to

```
cnt += (((class[i]).grade) == 'F');
```


The member `grade` of the *i*th element (counting from zero) of the array of structures `class` is selected. A test is made to see if it is equal to `'F'`. If equality holds, then the value of the expression

```
class[i].grade == 'F'
```

is 1, and the value of `cnt` is incremented. If equality does not hold, then the value of the expression is 0, and the value of `cnt` remains unchanged.

■
```
return cnt;
```

The number of failing grades is returned to the calling environment.

C provides the member access operator -> to access the members of a structure via a pointer. (This operator is typed on the keyboard as a minus sign followed by a greater than sign.) If a pointer variable is assigned the address of a structure, then a member of the structure can be accessed by a construct of the form

pointer_to_structure -> *member_name*

A construct that is equivalent to this is

(**pointer_to_structure*).*member_name*

The parentheses are necessary. Along with () and [], the operators "." and -> have the highest precedence and associate from left to right. Thus, the preceding construct without parentheses would be equivalent to

**(pointer_to_structure.member_name)*

This is an error because only a structure can be used with the "." operator, not a pointer to a structure.

Let us illustrate the use of -> by writing a function that adds complex numbers. First, suppose we have the following `typedef` in a header file:

In file complex.h

```
struct complex {
   double   re;      /* real part */
   double   im;      /* imag part */
};
```

```
typedef   struct complex   complex;
```

Then in another file we write

In file 2_add.c

```
#include "complex.h"

void add(complex *a, complex *b, complex *c)      /* a = b + c */
{
   a -> re = b -> re + c -> re;
   a -> im = b -> im + c -> im;
}
```

Note that a, b, and c are pointers to structures. If we think of the pointers as representing complex numbers, then we are adding b and c and putting the result into a. That is the sense of the comment that follows the header to the function definition.

In the following table, we have illustrated some of the ways that the two member access operators can be used. In the table, we assume that the user-defined type `struct student`, which we presented earlier, has already been declared.

Declarations and assignments

`struct student   tmp, *p = &tmp;`
`tmp.grade = 'A';`
`tmp.last_name = "Casanova";`
`tmp.student_id = 910017;`

Expression	Equivalent expression	Conceptual value
`tmp.grade`	`p -> grade`	A
`tmp.last_name`	`p -> last_name`	Casanova
`(*p).student_id`	`p -> student_id`	910017
`* p -> last_name + 1`	`(*(p -> last_name)) + 1`	D
`*(p -> last_name + 2)`	`(p -> last_name)[2]`	s

9.3 Operator Precedence and Associativity: A Final Look

We now want to display the entire precedence and associativity table for all the C operators. The operators "." and -> have been introduced in this chapter. These operators, together with () and [], have the highest precedence.

Operators	Associativity
() [] . -> ++ (*postfix*) -- (*postfix*)	left to right
++ (*prefix*) -- (*prefix*) ! ~ sizeof (*type*) + (*unary*) - (*unary*) & (*address*) * (*dereference*)	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= = %= >>= <<= &= ^= |=	right to left
, (*comma operator*)	left to right

Although the complete table of operators is extensive, some simple rules apply. The primary operators are function parentheses, subscripting, and the two member access operators. These four operators are of highest precedence. Unary operators come next, followed by the arithmetic operators. Arithmetic operators follow the usual convention; namely, multiplicative operators have higher precedence than additive operators. Assignments of all kinds are of lowest precedence, with the exception of the still more lowly comma operator. If a programmer does not know the rules of precedence and

associativity in a particular situation, he or she should either look up the rules or use parentheses. If you work at a particular location on a regular basis, you should consider copying this table and pasting it on the wall right next to where you work.

9.4 Using Structures with Functions

In C, structures can be passed as arguments to functions and can be returned from them. When a structure is passed as an argument to a function, it is passed by value, meaning that a local copy is made for use in the body of the function. If a member of the structure is an array, then the array gets copied as well. If the structure has many members, or members that are large arrays, then passing the structure as an argument can be relatively inefficient. For most applications, an alternate scheme is to write functions that take an address of the structure as an argument instead.

Business applications often use structures that have lots of members. Let us imagine that we have such a structure:

```
typedef struct {
   char                  name[25];
   int                   employee_id;
   struct dept           department;
   struct home_address   *a_ptr;
   double                salary;
   .....
} employee_data;
```

Notice that the `department` member is itself a structure. Because the compiler has to know the size of each member, the declaration for `struct dept` has to come first. Let us suppose that it was given by

```
struct dept {
   char   dept_name[25];
   int    dept_no;
};
```

Notice that the `a_ptr` member in the type definition of `employee_data` is a pointer to a structure. Because the compiler already knows the size of a pointer, this structure need not be defined first.

Now, suppose we want to write a function to update employee information. There are two ways we can proceed. The first way is as follows:


```
employee_data update(employee_data e)
{
   .....
   printf("Input the department number:  ");
   scanf("%d", &n);
   e.department.dept_no = n;
   .....
   return e;
}
```

Notice that we are accessing a member of a structure within a structure, because

`e.department.dept_no` is equivalent to `(e.department).dept_no`

To use the function `update()`, we could write in `main()` or in some other function

```
employee_data   e;
e = update(e);
```

Here, `e` is being passed by value, causing a local copy of `e` to be used in the body of the function; when a structure is returned from `update()`, it is assigned to `e`, causing a member-by-member copy to be performed. Because the structure is large, the compiler must do a lot of copy work. An alternate way to proceed is to write

```
void update(employee_data *p)
{
   .....
   printf("Input the department number:  ");
   scanf("%d", &n);
   p -> department.dept_no = n;
   .....
```

In this example, the construct

`p -> department.dept_no` is equivalent to `(p -> department).dept_no`

This illustrates how a member of a structure within a structure can be accessed via a pointer. To use this version of the `update()` function, we could write in `main()` or in some other function

```
employee_data   e;
update(&e);
```

Here, the address of `e` is being passed, so no local copy of the structure is needed within the `update()` function. For most applications this is the more efficient of the two methods. (See exercise 5, on page 438, for further discussion.)

9.5 Initialization of Structures

All external and static variables, including structures, that are not explicitly initialized by the programmer are automatically initialized by the system to zero. In traditional C, only external and static variables can be initialized. ANSI C allows automatic variables, including structures, to be initialized as well.

The syntax for initializing structures is similar to that for initializing arrays. A structure variable in a declaration can be initialized by following it with an equal sign and a list of constants contained within braces. If not enough values are used to assign all the members of the structure, the remaining members are assigned the value zero by default. Some examples are

```
card   c = {13, 'h'};   /* the king of hearts */

complex   a[3][3] = {
   {{1.0, -0.1}, {2.0, 0.2}, {3.0, 0.3}},
   {{4.0, -0.4}, {5.0, 0.5}, {6.0, 0.6}},
};   /* a[2][] is assigned zeroes */

struct fruit   frt = {"plum", 150};

struct home_address {
   char   *street;
   char   *city_and_state;
   long   zip_code;
} address = {"87 West Street", "Aspen, Colorado", 80526};

struct home_address   previous_address = {0};
```

The last example illustrates a convenient way to initialize all members of a structure to have value zero. It causes pointer members to be initialized with the pointer value `NULL` and array members to have their elements initialized to zero.

9.6 An Example: Playing Poker

Let us use the concepts that we have presented thus far in this chapter to write a program to play poker. The program will compute the probability that a flush is dealt, meaning that all five cards in a hand are of the same suit. In the exercises, we will discuss ways of extending the program. *Caution:* We are going to redesign our `card` structure.

Our program consists of many small functions. The simplest scheme is to put them all in one file. We will present the functions one after another as they occur in the file. Where appropriate, we will discuss key ideas. Here is what occurs at the top of the file:

In file poker.c

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

#define   NDEALS     3000   /* number of deals */
#define   NPLAYERS   6      /* number of players */

typedef   enum {clubs, diamonds, hearts, spades}   cdhs;

struct card {
   int    pips;
   cdhs   suit;
};

typedef   struct card   card;
```

We are envisioning a game with six players. Because we are interested in the probability of a flush occurring, we want to deal many hands. Notice that the member `suit` is an enumeration type. As we will see, the use of the enumeration type makes the code very readable. Next in the file comes the function prototypes:

```
card   assign_values(int pips, cdhs suit);
void   prn_card_values(card *c_ptr);
void   play_poker(card deck[52]);
void   shuffle(card deck[52]);
void   swap(card *p, card *q);
void   deal_the_cards(card deck[52], card hand[NPLAYERS][5]);
int    is_flush(card h[5]);
```

We begin our program development with `main()`. Every time we need a new function, we write it at the bottom of the file and put its function prototype in the list just above `main()`.

```
int main(void)
{
   cdhs   suit;
   int    i, pips;
   card   deck[52];

   for (i = 0; i < 52; ++i) {
      pips = i % 13 + 1;
      if (i < 13)
       suit = clubs;
      else if (i < 26)
       suit = diamonds;
      else if (i < 39)
       suit = hearts;
      else
       suit = spades;
      deck[i] = assign_values(pips, suit);
   }
   for (i = 26; i < 39; ++i)        /* print out the hearts */
      prn_card_values(&deck[i]);
   play_poker(deck);
   return 0;
}
```

In `main()`, we first assign values to each element in the array `deck`. We think of these elements as playing cards in a deck. To check that the code works as expected, we print out the hearts.

```
card assign_values(int pips, cdhs suit)
{
   card   c;

   c.pips = pips;
   c.suit = suit;
   return c;
}
```

Notice that the identifiers `pips` and `suit` are parameters in the function definition as well as members of the structure `c`. This sort of dual use of names is common.


```
void prn_card_values(card *c_ptr)
{
   int    pips = c_ptr -> pips;
   cdhs   suit = c_ptr -> suit;
   char   *suit_name;

   if (suit == clubs)
      suit_name = "clubs";
   else if (suit == diamonds)
      suit_name = "diamonds";
   else if (suit == hearts)
      suit_name = "hearts";
   else if (suit == spades)
      suit_name = "spades";
   printf("card: %2d of %s\n", pips, suit_name);
}
```

In the function `prn_card_values()`, we use `pips` and `suit` as local variables. The compiler cannot confuse these identifiers with the members of `card` because members of a structure can be accessed only via the "." and -> operators.

Our next function is `play_poker()`. Because it is central to this program and contains some crucial ideas, we will look at it in detail.

```
void play_poker(card deck[52])
{
   int    flush_cnt = 0, hand_cnt = 0;
   int    i, j;
   card   hand[NPLAYERS][5];   /* each player dealt 5 cards */

   srand(time(NULL));       /* seed random-number generator */
   for (i = 0; i < NDEALS; ++i) {
      shuffle(deck);
      deal_the_cards(deck, hand);
      for (j = 0; j < NPLAYERS; ++j) {
         ++hand_cnt;
         if (is_flush(hand[j])) {
            ++flush_cnt;
            printf("%s%d\n%s%d\n%s%f\n\n",
               "      Hand number:  ", hand_cnt,
               "     Flush number:  ", flush_cnt,
               "Flush probability:  ",
               (double) flush_cnt / hand_cnt);
         }
      }
   }
}
```

Dissection of the play_poker() Function

```
card   hand[NPLAYERS][5];   /* each player dealt 5 cards */
```

The identifier hand is a two-dimensional array. It can be viewed also as an array of arrays. Because the symbolic constant NPLAYERS has value 6, we can think of hand as six hands, each with five cards.

```
srand(time(NULL));          /* seed random-number generator */
```

The function `srand()` is used to seed the random-number generator invoked by `rand()`. The value supplied as an argument to `srand()` is called the seed. Here, we have used for the seed the value returned by the function call `time(NULL)`. This function is in the standard library, and its function prototype is in the header file *time.h*. Because `time()` produces an integer value that is obtained from the internal clock, each time we run the program a different seed is produced, causing different numbers to be generated by `rand()`. Hence, every time we run the program different hands are dealt.

```
for (i = 0; i < NDEALS; ++i) {
   shuffle(deck);
   deal_the_cards(deck, hand);
```

Before dealing the cards, we shuffle the deck. As we will see, this is where we use `rand()` to simulate the shuffling.

```
for (j = 0; j < NPLAYERS; ++j) {
   ++hand_cnt;
   if (is_flush(hand[j])) {
      ++flush_cnt;
      .....
```

After the cards are dealt, we check each player's hand to see if it is a flush. If it is, then we increment `flush_cnt` and print.

The function that shuffles the cards comes next. It is quite simple. We go through the deck swapping each card with another card that is chosen randomly. Note that the 52 in the header to the function definition is only for the benefit of the human reader. The compiler disregards it.


```
void shuffle(card deck[52])
{
   int   i, j;

   for (i = 0; i < 52; ++i) {
      j = rand() % 52;
      swap(&deck[i], &deck[j]);
   }
}

void swap(card *p, card *q)
{
   card   tmp;

   tmp = *p;
   *p = *q;
   *q = tmp;
}
```

Even though the `swap()` function is used to swap structures, its general form is the same.

```
void deal_the_cards(card deck[52], card hand[NPLAYERS][5])
{
   int   card_cnt = 0, i, j;

   for (j = 0; j < 5; ++j)
      for (i = 0; i < NPLAYERS; ++i)
         hand[i][j] = deck[card_cnt++];
}
```

Because the compiler disregards the first size in an array in a parameter declaration, an equivalent header for this function definition is

```
void deal_the_cards(card deck[], card hand[][5])
```

Also, because arrays in parameter declarations are really pointers, another equivalent header is given by

```
void deal_the_cards(card *deck, card (*hand)[5])
```

In each of these headers, the 5 is needed by the compiler to generate the correct storage mapping function. Notice that in the function the cards are dealt one at a time to each of the hands in turn. In contrast, the code

```
for (i = 0; i < NPLAYERS; ++i)
   for (j = 0; j < 5; ++j)
      hand[i][j] = deck[cnt++];
```

would have the effect of dealing five cards all at once to each of the hands. Because we are trying to simulate a poker game as it is actually played, the code that we used in `deal_the_cards()` is preferable.

```
int is_flush(card h[5])
{
   int   i;

   for (i = 1; i < 5; ++i)
      if (h[i].suit != h[0].suit)
         return 0;
   return 1;
}
```

The function `is_flush()` checks to see if the cards in a hand are all of the same suit. It is invoked in the function `play_poker()` with the expression `is_flush(hand[j])`. Recall that `hand` is an array of arrays, making `hand[j]` itself an array of type `card`. When `is_flush()` is called, the elements of this array are accessed.

9.7 Unions

A union, like a structure, is a derived type. Unions follow the same syntax as structures but have members that share storage. A union type defines a set of alternative values that may be stored in a shared portion of memory. The programmer is responsible for interpreting the stored values correctly. Consider the declaration

```
union int_or_float {
   int     i;
   float   f;
};
```

In this declaration, `union` is a keyword, `int_or_float` is the union tag name, and the variables `i` and `f` are members of the union. This declaration creates the derived data type `union int_or_float`. The declaration can be thought of as a template; it creates the type, but no storage is allocated. The tag name, along with the keyword `union`, can now be used to declare variables of this type.


```
union int_or_float   a, b, c;
```

This declaration allocates storage for the identifiers `a`, `b`, and `c`. For each variable the compiler allocates a piece of storage that can accommodate the largest of the specified members. The notation used to access a member of a union is identical to that used to access a member of a structure.

The following example shows how what is stored in a union can be interpreted in different ways:

In file numbers.c

```
typedef union int_or_float {
   int     i;
   float   f;
} number;

int main(void)
{
   number   n;

   n.i = 4444;
   printf("i: %10d     f: %16.10e\n", n.i, n.f);
   n.f = 4444.0;
   printf("i: %10d     f: %16.10e\n", n.i, n.f);
   return 0;
}
```

This little experiment demonstrates how your system overlays an `int` and a `float`. The output of this program is system-dependent. Here is what is printed on our system:

```
i:       4444     f: 6.227370375e-41
i: 1166729216     f: 4.4440000000e+03
```

Dissection of the *int_or_float* Program

```
typedef union int_or_float {
   int     i;
   float   f;
} number;
```

As with structures, the `typedef` mechanism can be used to abbreviate long type names.

■ `number n;`

The type `number` is equivalent to `union int_or_float`. This declaration causes the system to allocate space for the union variable `n`. That space must accommodate the larger of the two members of `n`. If we assume that an `int` is stored in 2 or 4 bytes and that a `float` is stored in 4 bytes, then the system will allocate 4 bytes for `n`.

■
```
n.i = 4444;
printf("i: %10d      f: %16.10e\n", n.i, n.f);
```

The `int` member `n.i` is assigned the `int` value 4444. When we print the value of `n.i` in the format of a decimal integer, 4444, of course, is printed. However, the same piece of memory is interpreted as a float when we print the member `n.f` in the `%e` format. In this case, something radically different from 4444 is printed.

■
```
n.f = 4444.0;
printf("i: %10d      f: %16.10e\n", n.i, n.f);
```

Now we are playing the game the other way. The `float` member `n.f` is assigned the value 4444.0. When `n.f` is printed in the `%e` format, we get the correct value, but when we print the underlying storage in the `%d` format, we get something quite different.

The point is that the system will interpret the same stored values according to which member component is selected. It is the programmer's responsibility to choose the right one.

Unions are used in applications that require multiple interpretations for a given piece of memory. But more commonly, they are used to conserve storage by allowing the same space in memory to be used for a variety of types. The members of a union can be structures or other unions. The following is an example:

```
struct flower {
   char                          *name;
   enum {red, white, blue}       color;
};

struct fruit {
   char    *name;
   int     calories;
};
```



```
struct vegetable {
   char    *name;
   int     calories;
   int     cooking_time;      /* in minutes */
};

union flower_fruit_or_vegetable {
   struct flower       flw;
   struct fruit        frt;
   struct vegetable    veg;
};

union flower_fruit_or_vegetable    ffv;
```

Having made these declarations, we can use a statement such as

```
ffv.veg.cooking_time = 7;
```

to assign a value to the member `cooking_time` of the member `veg` in the union `ffv`.

9.8 Bit Fields

An `int` or `unsigned` member of a structure or union can be declared to consist of a specified number of bits. Such a member is called a *bit field*, and the number of associated bits is called its *width*. The width is specified by a nonnegative constant integral expression following a colon. The width is at most the number of bits in a machine word. Typically, bit fields are declared as consecutive members of a structure, and the compiler packs them into a minimal number of machine words. An example is

```
struct pcard {              /* a packed representation */
   unsigned   pips : 4;
   unsigned   suit : 2;
};
```

A variable of type `struct pcard` has a 4-bit field called `pips` that is capable of storing the 16 values 0 to 15, and a 2-bit field called `suit` that is capable of storing the values 0, 1, 2, and 3, which can be used to represent clubs, diamonds, hearts, and spades, respectively. Thus, the thirteen `pips` values and the four `suit` values needed to represent playing cards can be represented compactly with 6 bits. Suppose that we make the declaration

```
struct pcard   c;
```

To assign to c the nine of diamonds, we can write

```
c.pips = 9;
c.suit = 1;
```

The syntax for a bit field within a structure or union is given by

bit_field_member ::= { int | unsigned }$_1$ { *identifier* }$_{opt}$: *expr*
expr ::= *constant_integral_expression*

Whether the compiler assigns the bits in left-to-right or right-to-left order is machine-dependent. Although some machines assign bit fields across word boundaries, most of them do not. Thus, on a machine with 4-byte words, the declaration

```
struct abc {
   int   a : 1, b : 16, c : 16;
} x;
```

typically would cause x to be stored in two words, with a and b stored in the first word and c stored in the second. Only nonnegative values can be stored in unsigned bit fields. For int bit fields, what happens is system-dependent. On some systems the high-order bit in the field is treated as the sign bit. (See exercise 25, on page 443.) In most applications, unsigned bit fields are used.

The chief reason for using bit fields is to conserve memory. On machines with 4-byte words, we can store 32 1-bit variables in a single word. Alternatively, we could use 32 char variables. The amount of memory saved by using bit fields can be substantial.

There are some restrictions, however. Arrays of bit fields are not allowed. Also, the address operator & cannot be applied to bit fields. This means that a pointer cannot be used to address a bit field directly, although use of the member access operator –> is acceptable.

One last point: Unnamed bit fields can be used for padding and alignment purposes. Suppose our machine has 4-byte words, and suppose we want to have a structure that contains six 7-bit fields with three of the bit fields in the first word and three in second. The following declaration accomplishes this:

```
struct small_integers {
   unsigned   i1 : 7, i2 : 7, i3 : 7,
                 : 11,                    /* align to next word */
              i4 : 7, i5 : 7, i6 : 7;
};
```


Another way to cause alignment to a next word is to use an unnamed bit field with zero width. Consider the code

```
struct abc {
   unsigned   a : 1, : 0, b : 1, : 0, c : 1;
};
```

This creates three 1-bit fields in three separate words.

9.9 An Example: Accessing Bits and Bytes

In this section we give an example of how the bits and bytes of a word in memory can be accessed. One way we can do this is through the use of bitwise operators and masks, as explained in Section 7.1, "Bitwise Operators and Expressions," on page 331 and Section 7.2, "Masks," on page 337. Here, we illustrate how it can be done using bit fields. Our program will be for machines having 4-byte words.

In file check_bits.c

```
#include <stdio.h>

typedef struct {
   unsigned   b0 : 8, b1 : 8, b2 : 8, b3 : 8;
} word_bytes;

typedef struct {
   unsigned
      b0 : 1, b1  : 1, b2  : 1, b3  : 1, b4  : 1, b5  : 1, b6  : 1,
      b7 : 1, b8  : 1, b9  : 1, b10 : 1, b11 : 1, b12 : 1, b13 : 1,
      b14: 1, b15 : 1, b16 : 1, b17 : 1, b18 : 1, b19 : 1, b20 : 1,
      b21: 1, b22 : 1, b23 : 1, b24 : 1, b25 : 1, b26 : 1, b27 : 1,
      b28: 1, b29 : 1, b30 : 1, b31;
} word_bits;

typedef union {
   int          i;
   word_bits    bit;
   word_bytes   byte;
} word;
```

```
int main(void)
{
   word   w = {0};
   void   bit_print(int);

   w.bit.b8 = 1;
   w.byte.b0 = 'a';
   printf("w.i = %d\n", w.i);
   bit_print(w.i);
   return 0;
}
```

The code in `main()` shows how a particular bit or byte in a word can be accessed. We use the function `bit_print()`, which we presented in Section 7.3, "Software Tools: Printing an `int` Bitwise," on page 338, to see precisely what is happening to the word in memory. Because machines vary on whether bits and bytes are counted from the high-order or low-order end of a word, the use of a utility such as `bit_print()` is essential. On one machine, our program caused the following to be printed:

```
w.i = 353
00000000 00000000 00000001 01100001
```

whereas on another machine we obtained

```
w.i = 1635778560
01100001 10000000 00000000 00000000
```

Because machines vary with respect to word size and with respect to how bits and bytes are counted, code that uses bit fields may not be portable.

9.10 The ADT Stack

The term *abstract data type* (ADT) is used in computer science to mean a data structure together with its operations, without specifying an implementation. Suppose we wanted a new integer type, one that could hold arbitrarily large values. The new integer type together with its arithmetic operations is an ADT. It is up to each individual system to determine how the values of integer data types are represented and manipulated computationally. Native types such as `char`, `int`, and `double` are implemented by the C compiler.

Programmer-defined types are frequently implemented with structures. In this section, we develop and implement the ADT *stack*, one of the most useful standard data structures. A stack is a data structure that allows insertion and deletion of data to occur only at a single restricted element, the top of the stack. This is the *last-in-first-out* (LIFO) discipline. Conceptually, a stack behaves like a pile of trays that pops up or is pushed down when trays are removed or added. The typical operations that can be used with a stack are *push*, *pop*, *top*, *empty*, *full*, and *reset*. The push operator places a value on the stack. The pop operator retrieves and deletes a value off the stack. The top operator returns the top value from the stack. The empty operator tests if th*e stack is empty. The full operator tests if the stack is full. The reset operator clears the stack, or initializes it. The stack, along with these operations, is a typical ADT.

We will use a fixed-length `char` array to store the contents of the stack. (Other implementation choices are possible; in Section 10.5, "Stacks," on page 461, we will implement a stack as a linked list.) The top of the stack will be an integer-valued member named `top`. The various stack operations will be implemented as functions, each of whose parameter lists includes a parameter of type pointer to `stack`. By using a pointer, we avoid copying a potentially large stack to perform a simple operation.

In file stack.c

```
/* An implementation of type stack. */

#define   MAX_LEN    1000
#define   EMPTY      -1
#define   FULL       (MAX_LEN - 1)

typedef   enum boolean {false, true}   boolean;

typedef struct stack {
   char   s[MAX_LEN];
   int    top;
} stack;

void reset(stack *stk)
{
   stk -> top = EMPTY;
}

void push(char c, stack *stk)
{
   stk -> top++;
   stk -> s[stk -> top] = c;
}
```

```
char pop(stack *stk)
{
   return (stk -> s[stk -> top--]);
}

char top(const stack *stk)
{
   return (stk -> s[stk -> top]);
}

boolean empty(const stack *stk)
{
   return ((boolean) (stk -> top == EMPTY));
}

boolean full(const stack *stk)
{
   return ((boolean) (stk -> top == FULL));
}
```

Dissection of the *stack* Implementation

- `typedef enum boolean {false, true} boolean;`

We use this `typedef` to give a new name, `boolean`, to the enumeration type, `enum boolean`. Note that the new name coincides with the tag name. Programmers often do this.

-
```
typedef struct stack {
   char   s[MAX_LEN];
   int    top;
} stack;
```

This code declares the structure type `struct stack`, and at the same time uses a `typedef` to give `struct stack` the new name `stack`. Here is an equivalent way to write this:

```
struct stack {
   char   s[MAX_LEN];
   int    top;
};

typedef   struct stack   stack;
```


The structure has two members, the array member `s` and the `int` member `top`.

-
```
void reset(stack *stk)
{
   stk -> top = EMPTY;
}
```

The member `top` in the stack pointed to by `stk` is assigned the value `EMPTY`. This conceptually resets the stack, making it empty. In the calling environment, if `st` is a stack, we can write

```
reset(&st);
```

to reset `st` or initialize it. At the beginning of a program, we usually start with an empty stack.

-
```
void push(char c, stack *stk)
{
   stk -> top++;
   stk -> s[stk -> top] = c;
}
```

-
```
char pop(stack* stk)
{
   return (stk -> s[stk -> top--]);
}
```

The operation *push* is implemented as a function of two arguments. First, the member `top` is incremented. Note that

`stk -> top++` is equivalent to `(stk -> top)++`

Then the value of `c` is shoved onto the top of the stack. This function assumes that the stack is not full. The operation *pop* is implemented in like fashion. It assumes the stack is not empty. The value of the expression

```
stk -> top--
```

is the value that is currently stored in the member `top`. Suppose this value is 7. Then

```
stk -> s[7]
```

gets returned, and the stored value of `top` in memory gets decremented, making its value 6.

```
boolean empty(const stack *stk)
{
   return ((boolean) (stk -> top == EMPTY));
}

boolean full(const stack *stk)
{
   return ((boolean) (stk -> top == FULL));
}
```

Each of these functions tests the `stack` member `top` for an appropriate condition and returns a value of type `boolean`, either `true` or `false`. Suppose the expression

```
stk -> top == EMPTY
```

in the body of `empty()` is true. Then the expression has the `int` value 1. This value gets cast to the type `boolean`, making it `true`, and that is what gets returned.

To test our stack implementation, we can put the preceding code in a *.c* file and add the following code at the bottom. Our function `main()` enters the characters of a string onto a stack and then pops them, printing each character out in turn. The effect is to print in reverse order the characters that were pushed onto the stack.

In file stack.c

```
/* Test the stack implementation by reversing a string. */

#include <stdio.h>

int main(void)
{
   char    str[] = "My name is Laura Pohl!";
   int     i;
   stack   s;
   reset(&s);                          /* initialize the stack */
   printf(" In the string: %s\n", str);
   for (i = 0; str[i] != '\0'; ++i)
      if (!full(&s))
       push(str[i], &s);               /* push a char on the stack */
   printf("From the stack: ");
   while (!empty(&s))
      putchar(pop(&s));                /* pop a char off the stack */
   putchar('\n');
   return 0;
}
```


The output from this test program is

```
 In the string: My name is Laura Pohl!
From the stack: !lhoP arauL si eman yM
```

Note that the expression `&s`, the address of the `stack` variable `s`, is used as an argument whenever we call a stack function. Because each of these functions expects a pointer of type `stack *`, the expression `&s` is appropriate.

Summary

1 Structures and unions are principal methods by which the programmer can define new types.

2 The `typedef` facility can be used to give new names to types. Programmers routinely use `typedef` to give a new name to a structure or union type.

3 A structure is an aggregation of components that can be treated as a single variable. The components of the structure are called members.

4 Members of structures can be accessed by using the member access operator ".". If `s` is a structure variable with a member named `m`, then the expression `s.m` refers to the value of the member `m` within the structure `s`.

5 Members of structures can also be accessed by the member access operator `->`. If `p` is a pointer that has been assigned the value `&s`, then the expression `p -> m` also refers to `s.m`. Both "." and `->` have highest precedence among C operators.

6 In ANSI C, if `a` and `b` are two variables of the same structure type, then the assignment expression `a = b` is valid. It causes each member of `a` to be assigned the value of the corresponding member of `b`. Also, a structure expression can be passed as an argument to a function and returned from a function. (Many traditional C compilers also have these capabilities, but not all of them.)

7 When a structure variable is passed as an argument to a function, it is passed "call-by-value." If the structure has many members, or members that are large arrays, this may be an inefficient way of getting the job done. If we redesign the function

definition so that a pointer to the structure instead of the structure itself is used, then a local copy of the structure will not be created.

8 A union is like a structure, except that the members of a union share the same space in memory. Unions are used principally to conserve memory. The space allocated for a union variable can be used to hold a variety of types, specified by the members of the union. It is the programmer's responsibility to know which representation is currently stored in a union variable.

9 The members of structures and unions can be arrays or other structures and unions. Considerable complexity is possible when nesting arrays, structures, and unions within each other. Care must be taken that the proper variables are being accessed.

10 A bit field is an `int` or `unsigned` member of a structure or union that has a specified number of bits. The number of bits is given as an integral constant expression following a colon. This number is called the width of the bit field. The width is limited to the number of bits in a machine word. Consecutive bit fields in a structure are stored typically as contiguous bits in a machine word, provided that they fit.

11 Bit fields can be unnamed, in which case they are used for padding or word alignment purposes. The unnamed bit field of width 0 is special. It causes immediate alignment on the next word.

12 How bit fields are implemented is system-dependent. Hence, their use need not be portable. Nonetheless, bit fields have important uses.

13 The stack is an abstract data type (ADT) that has many uses, especially in computer science. An ADT can be implemented many different ways.

Exercises

1 In some situations a `typedef` can be replaced by a `#define`. Here is an example:

```
typedef   float   DOLLARS;

int main(void)
{
   DOLLARS   amount = 100.0, interest = 0.07 * amount;

   printf("DOLLARS = %.2f\n", amount + interest);
   return 0;
}
```

Execute this program so you understand its effects, then replace the `typedef` by

```
#define   DOLLARS   float
```

When you recompile the program and execute it, does it behave as it did before?

2 In some situations a `typedef` cannot be replaced by a `#define`. Consider the following program:

```
typedef   float   DOLLARS;

int main(void)
{
   DOLLARS   amount = 100.00,
             interest = 0.07 * amount;
   {                                                  /* new block */
      float   DOLLARS;
      DOLLARS = amount + interest;
      printf("DOLLARS = %.2f\n", DOLLARS);
   }
   return 0;
}
```

Execute this program so that you understand its effects. If the `typedef` is replaced by the line

```
#define   DOLLARS   float
```

3 The program in the previous exercise used a new block to illustrate a certain point. Because of this, the code was not very natural. In this exercise, we use a `typedef` again, and this time the code is quite straightforward.

```
typedef   char *   string;

int main(void)
{
   string   a[] = {"I", "like", "to", "fight,"},
            b[] = {"pinch,", "and", "bite."};

   printf("%s %s %s %s %s %s %s\n",
      a[0], a[1], a[2], a[3], b[0], b[1], b[2]);
   return 0;
}
```

Execute the program so that you understand its effects. If the `typedef` is replaced by the line

```
#define   string   char *
```

then the program will not compile. Explain why the `typedef` works, but the `#define` does not. To make the program work with the `#define`, a single character needs to be added. What is the character?

4 Write a function `add(a, b, c)`, where the variables `a`, `b`, and `c` are of type `matrix`. Write a program to test your function.

5 Speed is not everything, but it counts for a lot in the computing world. If a structure is large, it is more efficient to pass an address of the structure than it is to pass the structure itself. Does this hold true if the structure is small? After all, there is a cost for dealing with pointers, too. Suppose you have many complex numbers to multiply. How should you design your multiplication function? You could pass structures as arguments:

```
complex mult(complex b, complex c)
{
   complex a;

   a.re = b.re * c.re - b.im * c.im;
   a.im = b.im * c.re + b.re * c.im;
   return a;
}
```


But it may be faster to pass addresses:

```
void mult(complex *a, complex *b, complex *c)
{
   .....
```

Complete this version of the `mult()` function, and write a test program to see which version is faster.

6 The program in this exercise uses a `typedef` to name the type "pointer to a function that takes a double and returns a double." First, run the program so that you understand its effects.

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

#define   PI   3.14159

typedef   double   dbl;              /* create an abbreviation */
typedef   dbl      (*PFDD)(dbl);     /* ptr to fct taking dbl
                                        and returning a dbl */

int main(void)
{
   PFDD   f = sin, g = cos;

   printf("f(%f) = %f\n", PI, f(PI));
   printf("g(%f) = %f\n", PI, g(PI));
   return 0;
}
```

Now modify the code so that the second `typedef` is replaced by

```
typedef   dbl   FDD(dbl);   /* fct of dbl returning a dbl */
typedef   FDD   *PFDD;      /* ptr to FDD */
```

Does the program still run? (It should.) Carefully explain how the `typedef` for `PFDD` works.

7 Write a function to do complex subtraction. Write two versions: one that returns a pointer to `complex` and one that returns a value of type `complex`.

8 Write a small collection of functions that perform operations on complex vectors and matrices. Include in your collection functions that print complex vectors and matrices on the screen. Write a program to test your functions.

9 Write a program that is able to produce a balanced meal. Start by defining a structure that contains the name of a food, its calories per serving, its food type such as meat or fruit, and its costs. The foods should be stored as an array of structures. The program should construct a meal that comes from four different food types and that meets calorie and cost constraints. The program should be capable of producing a large number of different menus.

10 Create a structure that can describe a restaurant. It should have members that include the name, address, average cost, and type of food. Write a routine that prints out all restaurants of a given food type in order of cost, with the least costly first.

11 Put a list of student names, identification numbers, and grades into a file called *data*. For example, the beginning of the file might look like

```
Casanova    910017    A
Smith       934422    C
Jones       878766    B
.....
```

Write a program called *reorder* that can be used to read the data in the file and put it into the array `class` of type `struct student`. This can be done with redirection

reorder < data

The program should print out an ordered list of students and grades. Students with A grades should be listed first, students with B grades next, and so forth. Among all students having the same grade, the students should be listed alphabetically by name.

12 Modify the program that you wrote in the previous exercise by adding the function `class_average()` to compute the class average and print it. Assume that an A grade has value 4, a B grade has value 3, and so forth.

13 Experiment with the *poker* program. After dealing many hands, the last computed probability of getting a flush should approximate the mathematical probability. What is the mathematical probability of getting a flush? If you are dealt one card, then it will be of a certain suit, say hearts. What is the chance that the second card you are dealt will be a heart? Well, there are 51 cards left in the deck and 12 of them are hearts, so the probability is 12/51. What is the chance that the third card you are dealt will also be a heart? Because there are now 50 cards left in the deck and 11 of them are hearts, the probability that the second and third cards are hearts is

$(12 / 51) \times (11 / 50)$. Continuing in this fashion, we see that the mathematical probability of getting a flush is

$$\frac{12}{51} \times \frac{11}{50} \times \frac{10}{49} \times \frac{9}{48}$$

This product is approximately 0.00198. (Use your hand calculator to check that this is so.) When your machine plays poker, some of the computed probabilities should be larger than this number and some smaller. Is that the case?

14 Modify the *poker* program by adding the function `is_straight()`, which tests whether a poker hand is a straight. Whenever a straight is dealt, print the computed probability. Because a flush beats a straight, we expect the probability of a flush to be lower than the probability of a straight. Does your *poker* program confirm this?

15 Modify the *poker* program by adding the function `is_fullhouse()`, which tests whether a poker hand is a full house. Whenever a full house is dealt, print the computed probability. Because a full house beats a flush, we expect the probability of a full house to be lower than that of a flush. Does your *poker* program confirm this?

16 The experienced poker player arranges the hand to reflect its values. Write a program that arranges and prints out a hand of five cards in sorted order by pips value. Assume that an ace is highest in value, a king is next highest in value, and so forth.

17 (Advanced) Write a function `hand_value()`, which returns the poker value of the hand. The best possible hand is a straight flush, and the worst possible hand is no pair. Write another function that compares two hands to see which is best.

18 Consider the following version of the function `assign_values()`:

```
void assign_values(card c, int pips, cdhs suit)
{
   c.pips = pips;
   c.suit = suit;
}
```

When we test this function with the following code we find that it does not work as expected. Explain.

```
card   c;

assign_values(c, 13, diamonds);   /* the king of diamonds */
prn_card_values(&c);
```

19 Consider the following `union` declaration:

```
union a {
   int     a;
   char    b;
   float   a;
} a, b, c;
```

There is only one thing wrong. What is it?

20 Write a `typedef` for a structure that has two members. One member should be a union of `double` and `complex`, and the other member should be a "flag" that tells which domain is being used. Write functions that can add and multiply over both domains. Your functions should decide on appropriate conversions when mixing arguments from both domains.

21 In commercial applications, it is sometimes useful to use binary coded decimal codes (BCD), where 4 bits are used to represent a decimal digit. A 32-bit word can be used to represent an 8-digit decimal number. Use bit fields to implement this code. Write two conversion routines, one from binary to BCD and the second from BCD to binary.

22 In Section 4.10, "An Example: Boolean Variables," on page 170, we wrote a program that printed a table of values for some boolean functions. Rewrite that program using bit fields. Represent each boolean variable as a 1-bit field.

23 (Sieve of Eratosthenes) Use an array of structures, where each structure contains a word that is divided into bit fields, to implement the sieve algorithm for primes. On a machine with 4-byte words, we can use an array of 1,000 elements to find all primes less than 32,000. Let each bit represent an integer, and initialize all bits to zero. The idea is to cross out all the multiples of a given prime. A bit that is 1 will represent a composite number. Start with the prime 2. Every second bit is made 1 starting with bit 4. Then bit 3 is still 0. Every third bit is made 1, starting with bit 6. Bit 4 is 1, so it is skipped. Bit 5 is the next bit with value 0. Every fifth bit is made 1, starting with bit 10. At the end of this process, only those bits that are still zero represent primes.

24 What gets printed by the following program? Explain.

```
#include <stdio.h>

struct test {
   unsigned   a : 1, b : 2, c : 3;
};

int main(void)
{
   int           i;
   struct test   x;

   for (i = 0; i < 23; ++i) {
      x.a = x.b = x.c = i;
      printf("x.a = %d   x.b = %d   x.c = %d\n", x.a, x.b, x.c);
   }
   return 0;
}
```

What happens if you replace

```
x.a = x.b = x.c = i;   by   x.c = x.b = x.a = i;
```

25 Does your system implement signed arithmetic for `int` bit fields? Try the following code:

```
struct test {
   int   a : 3, b : 4;
} x = {0};

for ( ; ; )
   printf("x.a = %2d   x.b = %2d\n", x.a++, x.b++);
```

26 What gets printed? Explain.

```
typedef struct a { unsigned a : 1, : 0, b : 1; } a;
typedef struct b { unsigned a : 1, b : 1; } b;

printf("%.1f\n", (float) sizeof(a) / sizeof(b));
```

27 In this exercise, we want to consider = and == with respect to their use with structures. Suppose a and b are two structure variables of the same type. Then the expression a = b is valid, but the expression a == b is not. The operands of the == operator cannot be structures. (Beginning programmers often overlook this subtle point.) Write a small test program to see what your compiler does if you try to use the expression a == b, where a and b are structures of the same type.

28 (Project) The double-ended queue is a very useful ADT. To implement this ADT, start with the following constructs:

```
struct deque {        /* double ended queue */
   .....
};

typedef   struct deque          deque;
typedef   enum {false, true}    boolean;
typedef   data                  char;
```

The programmer-defined type deque implements a double-ended queue with an array. If you want the data stored in the queue to be ints, use

```
typedef   data                  int;
```

Function prototypes of useful operations include

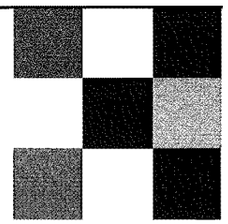
```
void      add_to_front(deque *, data);
void      add_to_rear(deque *, data);
data      take_from_front(deque *);
data      take_from_rear(deque *);
boolean   empty(deque *);
boolean   full(deque *);
```

Write and test an implementation of the double-ended queue ADT. (In Section 10.7, "Queues," on page 471, we will see how to do this with a linked list.)

29 Although the program in this exercise is system-dependent, it will run correctly on many different systems. See if it runs correctly on your machine.

```
/* The mystery program. */

#if (VAX || PC)
   #define   HEX0   0x6c6c6548
   #define   HEX1   0x77202c6f
   #define   HEX2   0x646c726f
   #define   HEX3   0x00000a21
#else
   #define   HEX0   0x48656c6c
   #define   HEX1   0x6f2c2077
   #define   HEX2   0x6f726c64
   #define   HEX3   0x210a0000
#endif

typedef union {
   char   what[16];
   long   cipher[4];
} mystery;

int main(void)
{
   mystery   x;

   x.cipher[0] = HEX0;      /* put a hex on the mystery */
   x.cipher[1] = HEX1;
   x.cipher[2] = HEX2;
   x.cipher[3] = HEX3;
   printf("%s", x.what);
   return 0;
}
```

Explain how the mystery program works. If the phrase that is printed is unfamiliar to you, ask a programmer who has worked in C for a number of years what its significance is.



Chapter 10

Structures and List Processing

In this chapter, we explain self-referential structures. We define structures with pointer members that refer to the structure containing them. Such data structures are called *dynamic data structures.* Unlike arrays or simple variables that are normally allocated at block entry, dynamic data structures often require storage management routines to explicitly obtain and release memory.

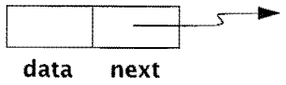
10.1 Self-referential Structures

Let us define a structure with a member field that points at the same structure type. We wish to do this in order to have an unspecified number of such structures nested together.

```
struct list {
   int           data;
   struct list   *next;
} a;
```

This declaration can be stored in two words of memory. The first word stores the member `data`, and the second word stores the member `next`. The pointer variable `next` is called a *link.* Each structure is linked to a succeeding structure by way of the member `next`. These structures are conveniently displayed pictorially with links shown as arrows.

A structure of type `struct list`



The pointer variable `next` contains either an address of the location in memory of the successor `list` element, or the special value `NULL` defined as 0. `NULL` is used to denote the end of the list. Let us see how all this works by manipulating

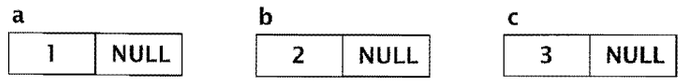
```
struct list   a, b, c;
```

We will perform some assignments on these structures:

```
a.data = 1;
b.data = 2;
c.data = 3;
a.next = b.next = c.next = NULL;
```

Pictorially, the result of this code is the following:

After assignment

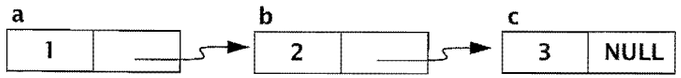


Let us chain these together.

```
a.next = &b;
b.next = &c;
```

These pointer assignments result in linking a to b to c.

After chaining



Now the links allow us to retrieve data from successive elements. Thus,

```
a.next -> data
```

has a value 2 and

```
a.next -> next -> data
```

has value 3.

10.2 Linear Linked Lists

A *linear linked list* is like a clothes line on which the data structures hang sequentially. A head pointer addresses the first element of the list, and each element points at a successor element, with the last element having a link value `NULL`. This discussion will use the following header file:

In file list.h

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

typedef   char   DATA;           /* will use char in examples */

struct linked_list {
   DATA                 d;
   struct linked_list   *next;
};

typedef   struct linked_list   ELEMENT;
typedef   ELEMENT              *LINK;
```

Storage Allocation

The specifications in *list.h* do not allocate storage. The system can allocate storage if we declare variables and arrays of type ELEMENT. But what makes these structures especially useful is that utility functions exist to allocate storage dynamically. The system provides `malloc()` in the standard library, and its prototype is in *stdlib.h*. A function call of the form

```
malloc(size)
```

returns a pointer to enough storage for an object of *size* bytes. The storage is not initialized. The argument to `malloc()` is of type `size_t`, and the value returned is of type pointer to `void`. Recall that `size_t` is the unsigned integer type that is defined in both *stddef.h* and *stdlib.h*. It is the type that results when the `sizeof` operator is used. If `head` is a variable of type `LINK`, then

```
head = malloc(sizeof(ELEMENT));
```

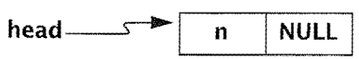
obtains a piece of memory from the system adequate to store an `ELEMENT` and assigns its address to the pointer `head`. As in the example, `malloc()` is used with the `sizeof` operator. A cast is unnecessary because `malloc()` returns a pointer to `void`, which can be assigned to a variable of any pointer type. The `sizeof` operator calculates the required number of bytes for the particular data structure.

In the code that follows, we will dynamically create a linear linked list storing the three characters *n*, *e*, and *w*. The code

```
head = malloc(sizeof(ELEMENT));
head -> d = 'n';
head -> next = NULL;
```

creates a single-element list.

Creating a linked list

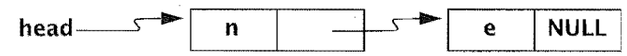


A second element is added by the assignments

```
head -> next = malloc(sizeof(ELEMENT));
head -> next -> d = 'e';
head -> next -> next = NULL;
```

It is now a two-element list.

A two-element linked list

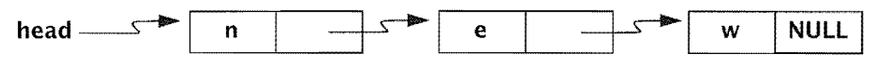


Finally, we add a last element:

```
head -> next -> next = malloc(sizeof(ELEMENT));
head -> next -> next -> d = 'w';
head -> next -> next -> next = NULL;
```

We have a three-element list pointed to by `head`, and the list ends when *next* has the sentinel value `NULL`.

A three-element linked list



10.3 List Operations

The following list includes some of the basic linear list operations:

Linear List Operations

1. Creating a list
2. Counting the elements
3. Looking up an element
4. Concatenating two lists
5. Inserting an element
6. Deleting an element

We will demonstrate the techniques for programming such operations on lists using both recursion and iteration. The use of recursive functions is natural because lists are a recursively defined construct. Each routine will require the specifications in file *list.h*. Observe that `d` in these examples could be redefined as an arbitrarily complicated data structure.

As a first example, we will write a function that will produce a list from a string. The function will return a pointer to the head of the resulting list. The heart of the function creates a list element by allocating storage and assigning member values.

```
/* List creation using recursion. */

#include <stdlib.h>
#include "list.h"

LINK string_to_list(char s[])
{
   LINK   head;

   if (s[0] == '\0')      /* base case */
      return NULL;
   else {
      head = malloc(sizeof(ELEMENT));
      head -> d = s[0];
      head -> next = string_to_list(s + 1);
      return head;
   }
}
```

Notice once more how recursion has a base case (the creation of the empty list) and a general case (the creation of the remainder of the list). The general recursive call returns as its value a LINK pointer to the remaining sublist.

Dissection of the string_to_list() Function

■
```
LINK string_to_list(char s[])
{
   LINK   head;
```

When a string is passed as an argument, a linked list of the characters in the string is created. Because a pointer to the head of the list will be returned, the type specifier in the header to this function definition is LINK.

■
```
if (s[0] == '\0')      /* base case */
   return (NULL);
```

When the end-of-string sentinel is detected, NULL is returned, and, as we will see, the recursion terminates. The value NULL is used to mark the end of the linked list.

■
```
else {
   head = malloc(sizeof(ELEMENT));
```

If the string s[] is not the null string, then malloc() is used to retrieve enough bytes to store an object of type ELEMENT. Because malloc() returns a pointer to void, it can be assigned to the variable head, which is a different pointer type. A cast is unnecessary. The pointer variable head now points at the block of storage provided by malloc().

■
```
head -> d = s[0];
```

The member d of the allocated ELEMENT is assigned the first character in the string s[].

■
```
head -> next = string_to_list(s + 1);
```

The pointer expression s + 1 points to the remainder of the string. The function is called recursively with s + 1 as an argument. The pointer member next is assigned the pointer value that is returned by string_to_list(s + 1). This recursive call returns as its value a LINK or, equivalently, a pointer to ELEMENT that points to the remaining sublist.

■
```
return head;
```

The function exits with the address of the head of the list.

This function can also be written as an iterative routine with the help of the additional auxiliary pointer tail. We will name the iterative version s_to_l() to distinguish it from the recursive version string_to_list().

In file iter_list.c

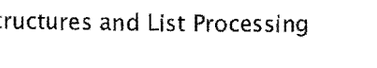
```
/* List creation using iteration. */

#include <stdlib.h>
#include "list.h"

LINK s_to_l(char s[])
{
   LINK    head = NULL, tail;
   int     i;
```

```
    if (s[0] != '\0') {                          /* first element */
       head = malloc(sizeof(ELEMENT));
       head -> d = s[0];
       tail = head;
       for (i = 1; s[i] != '\0'; ++i) {          /* add to tail */
        tail -> next = malloc(sizeof(ELEMENT));
        tail = tail -> next;
        tail -> d = s[i];
       }
       tail -> next = NULL;                      /* end of list */
    }
    return head;
}
```

Functions operating on lists often require local pointer variables such as `head` and `tail`. One should use such auxiliary pointers freely to simplify code. It is also important to hand-simulate these routines. It is useful to try your program on the empty list, the unit or single-element list, and the two-element list. Frequently, the empty list and the unit list are special cases.

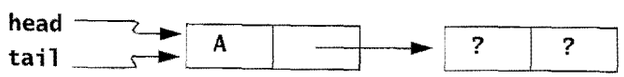
Passing a null string to `s_to_l()` creates the empty list by having the routine return with value `NULL`. Creating the first element is done by the first part of the code. The one-element list created from the string `"A"` is shown in the following diagram. This is the state of the computation before the member `next` is assigned the value `NULL`.

A one-element list



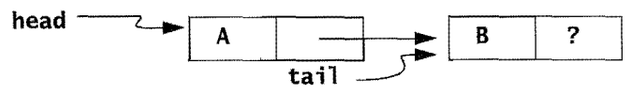
For the two-element case, say `"AB"`, list creation is as pictured. First, the one-element list containing `'A'` is created. The `for` statement is then executed, with `i` having value 1 and `s[1]` having value `'B'`. A new element is then allocated and attached to the list.

A second element is attached



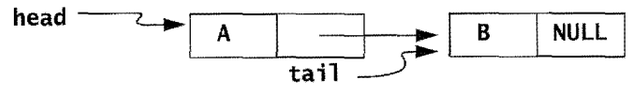
The statement `tail = tail -> next;` advances `tail` to the new element. Then its `d` member is assigned `'B'`.

Updating the tail



Now `s[2]` has value `\0` and the `for` statement is exited with a two-element list. Finally, the end of the list is marked with a `NULL`.

After assigning NULL



The undefined member values occur because `malloc()` is not required to initialize memory to zero.

10.4 Some List Processing Functions

We will write two additional recursive functions. The first counts the elements in a list, and the second prints the elements of a list. Both involve recurring down the list and terminating when the `NULL` pointer is found. All these functions use the header file *list.h*.

The function `count()` returns 0 if the list is empty; otherwise, it returns the number of elements in the list.

```
/* Count a list recursively. */

int count(LINK head)
{
   if (head == NULL)
      return 0;
   else
      return (1 + count(head -> next));
}
```

An iterative version of this function replaces the recursion with a for loop.

```
/* Count a list iteratively. */

int count_it(LINK head)
{
   int   cnt = 0;

   for ( ; head != NULL; head = head -> next)
      ++cnt;
   return cnt;
}
```

Keep in mind that head is passed "call-by-value," so that invoking count_it() does not destroy access to the list in the calling environment.

The routine print_list() recursively marches down a list printing the value of member variable d:

```
/* Print a list recursively. */

void print_list(LINK head)
{
   if (head == NULL)
      printf("NULL");
   else {
      printf("%c --> ", head -> d);
      print_list(head -> next);
   }
}
```

To illustrate the use of these functions, we will write a program that will convert the string "ABC" to a list and print it:

In file prn_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

LINK   string_to_list(char []);
void   print_list(LINK);
int    count(LINK);
```



```
int main(void)
{
   LINK   h;

   h = string_to_list("ABC");
   printf("The resulting list is\n");
   print_list(h);
   printf("\nThis list has %d elements.\n", count(h));
   return 0;
}
```

The program produces the following output:

```
The resulting list is
A --> B --> C --> NULL
This list has 3 elements.
```

Often one wishes to take two lists and return a single combined list. The concatenation of lists a and b, where a is assumed to be nonempty, will be the list b added to the end of list a. A function to concatenate will march down list a looking for its end, as marked by the null pointer. It will keep track of its last non-null pointer and will attach the b list to the next link in this last element of list a.

```
/* Concatenate list a and b with a as head. */

void concatenate(LINK a, LINK b)
{
   assert(a != NULL);
   if (a -> next == NULL)
      a -> next = b;
   else
      concatenate(a -> next, b);
}
```

Recursion allows us to avoid using any auxiliary pointers to march down the a list. In general, the self-referential character of list processing makes recursion natural to use. The form of these recursive functions is as follows:

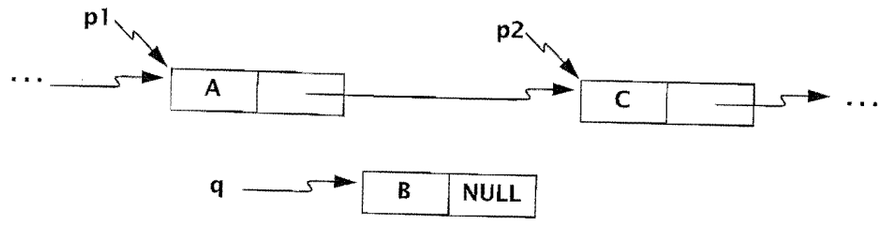
```
void generic_recursion(LINK head)
{
   if (head == NULL)
      do the base case
   else
      do the general case and recur with
   generic_recursion(head -> next)
}
```

Insertion

One of the most useful properties of lists is that insertion takes a fixed amount of time once the position in the list is found. In contrast, if one wished to place a value in a large array, retaining all other array values in the same sequential order, the insertion would take, on average, time proportional to the length of the array. The values of all elements of the array that come after the newly inserted value would have to be moved over one element.

Let us illustrate insertion into a list by having two adjacent elements pointed at by `p1` and `p2`, and inserting between them an element pointed at by `q`.

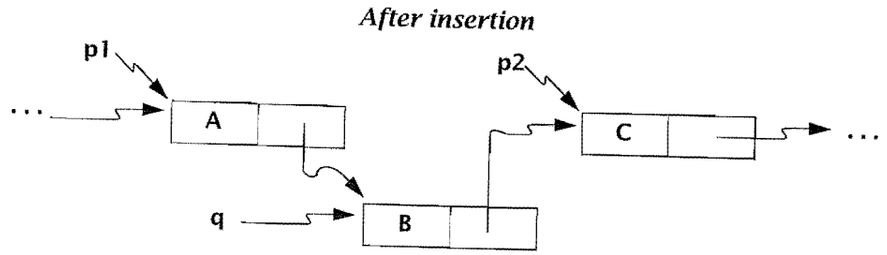
Before insertion



The following function `insert()` places the element pointed at by `q` between the elements pointed at by `p1` and `p2`:

```
/* Inserting an element in a linked list. */

void insert(LINK p1, LINK p2, LINK q)
{
   assert(p1 -> next == p2);
   p1 -> next = q;          /* insert */
   q -> next = p2;
}
```

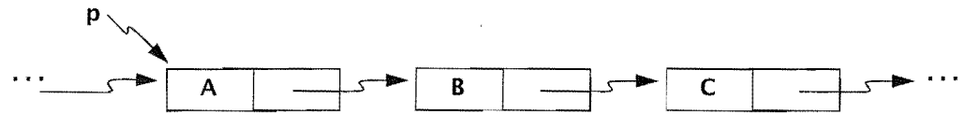
After insertion



Deletion

Deleting an element is very simple in a linked linear list. The predecessor of the element to be deleted has its link member assigned the address of the successor to the deleted element. Again, let us first illustrate the delete operation graphically. Here is the picture that we start with:

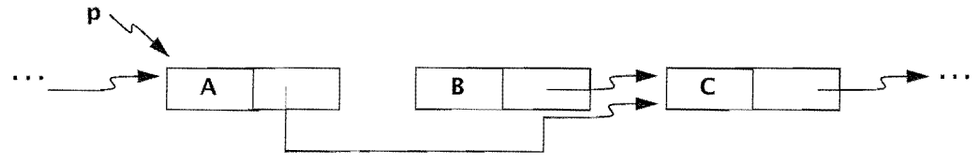
Before deletion



Now we execute the following code:

```
p -> next = p -> next -> next;
```

After deletion



As the diagram shows, the element containing `'B'` is no longer accessible and is of no use. Such an inaccessible element is called *garbage*. Because memory is frequently a critical resource, it is desirable that this storage be returned to the system for later use. This may be done with the *stdlib.h* function `free()`. When called as `free(p)`, previously allocated storage for the object pointed to by `p` is available to the system. The formal argument to `free()` is pointer to `void`.

Using `free()`, we can write a deletion routine that returns allocated list storage to the system.

```
/* Recursive deletion of a list. */

void delete_list(LINK head)
{
   if (head != NULL) {
      delete_list(head -> next);
      free(head);                    /* release storage */
   }
}
```

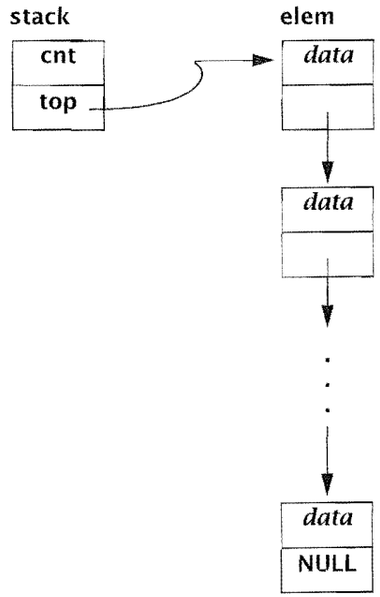
Because `free()` takes a single argument of type `void *`, we can pass a pointer of any type to `free()`. We do not have to use a cast because `void *` is the generic pointer type.

10.5 Stacks

We presented the abstract data type stack as an array in Section 9.10, "The ADT Stack," on page 430. Here, we will reimplement the ADT stack with a linear linked list. A stack has access restricted to the head of the list, which will be called its *top*. Furthermore, insertion and deletion occur only at the top, and under these restrictions the operations are known as *push* and *pop*, respectively.

A stack can be visualized as a pile of trays. A tray is always picked up from the top and a tray is always returned to the top. Graphically, stacks are drawn vertically.

A stack implementation



We will write a stack program that consists of a *.h* file and two *.c* files. Here is the header file:

In file stack.h

```
/* A linked list implementation of a stack. */

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

#define   EMPTY       0
#define   FULL        10000

typedef   char                      data;
typedef   enum {false, true}        boolean;

struct elem {                        /* an element on the stack */
   data          d;
   struct elem   *next;
};

typedef   struct elem   elem;

struct stack {
   int    cnt;                       /* a count of the elements */
   elem   *top;                      /* ptr to the top element */
};

typedef   struct stack   stack;

void      initialize(stack *stk);
void      push(data d, stack *stk);
data      pop(stack *stk);
data      top(stack *stk);
boolean   empty(const stack *stk);
boolean   full(const stack *stk);
```

The function prototypes of the six standard stack operations that we are going to implement are listed at the bottom of our header file. Conceptually, these functions behave like those we presented in Section 9.10, "The ADT Stack," on page 430. Here, our use of the type `data` makes our code reusable. (See Section 10.6, "An Example: Polish Notation and Stack Evaluation," on page 464, where we use the code again.)

In file stack.c

```
/* The basic stack routines. */

#include "stack.h"

void initialize(stack *stk)
{
   stk -> cnt = 0;
   stk -> top = NULL;
}

void push(data d, stack *stk)
{
   elem   *p;

   p = malloc(sizeof(elem));
   p -> d = d;
   p -> next = stk -> top;
   stk -> top = p;
   stk -> cnt++;
}

data pop(stack *stk)
{
   data   d;
   elem   *p;

   d = stk -> top -> d;
   p = stk -> top;
   stk -> top = stk -> top -> next;
   stk -> cnt--;
   free(p);
   return d;
}

data top(stack *stk)
{
   return (stk -> top -> d);
}

boolean empty(const stack *stk)
{
   return ((boolean) (stk -> cnt == EMPTY));
}
```



```
boolean full(const stack *stk)
{
   return ((boolean) (stk -> cnt == FULL));
}
```

The push() routine uses the storage allocator malloc() to create a new stack element, and the pop() routine returns the freed-up storage back to the system.

A stack is a *last-in-first-out* (LIFO) data structure. The last item to be pushed onto the stack is the first to be popped off. So if we were to push first 'a' and second 'b' onto a stack, then pop() would first pop 'b'. In main(), we use this property to print a string in reverse order. This serves as a test of our implementation of the ADT stack.

In file main.c

```
/* Test the stack implementation by reversing a string. */

#include "stack.h"

int main(void)
{
   char    str[] = "My name is Joanna Kelley!";
   int     i;
   stack   s;

   initialize(&s);                        /* initialize the stack */
   printf(" In the string: %s\n", str);
   for (i = 0; str[i] != '\0'; ++i)
      if (!full(&s))
         push(str[i], &s);                /* push a char on the stack */
   printf("From the stack: ");
   while (!empty(&s))
      putchar(pop(&s));                   /* pop a char off the stack */
   putchar('\n');
   return 0;
}
```

Observe that our function main() is very similar to what we wrote in Section 9.10, "The ADT Stack," on page 434. Although here we have implemented the ADT as a linked list and in Section 9.10, "The ADT Stack," on page 431, we implemented the ADT as a string, the use of the operations is similar. Here is the output of our program:

```
 In the string: My name is Joanna Kelley!
From the stack: !yelleK annaoJ si eman yM
```

10.6 An Example: Polish Notation and Stack Evaluation

Ordinary notation for writing expressions is called *infix*, where operators separate arguments. Another notation for expressions, one that is very useful for stack-oriented evaluation, is called *Polish*, or parenthesis-free, *notation*. In Polish notation, the operator comes after the arguments. Thus, for example,

`3, 7, +` is equivalent to the infix notation `3 + 7`

In Polish notation, going from left to right, the operator is executed as soon as it is encountered. Thus

`17, 5, 2, *, +` is equivalent to `17 + (5 * 2)`

A Polish expression can be evaluated by an algorithm using two stacks. The Polish stack contains the Polish expression, and the evaluation stack stores the intermediate values during execution. Here is a two-stack algorithm that evaluates Polish expressions where all operators are binary:

A two-stack algorithm to evaluate Polish expressions

1 If the Polish stack is empty, halt with the top of the evaluation stack as the answer.

2 If the Polish stack is not empty, pop the Polish stack into d. (We will use d, d1, and d2 to hold data.)

3 If d is a value, push d onto the evaluation stack.

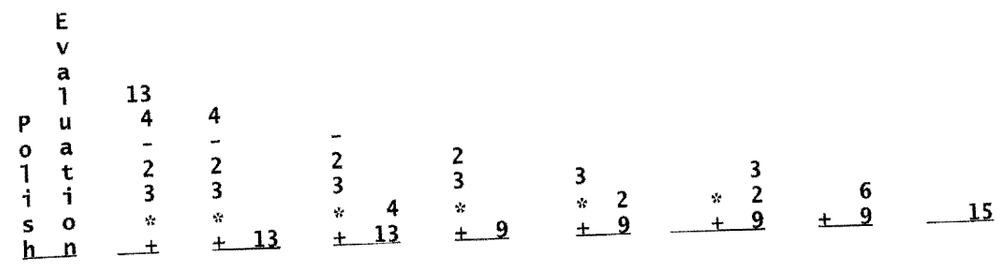
4 If d is an operator, pop the evaluation stack twice, first into d2 and then into d1. Compute d1 and d2 operated on by d, and push the result onto the evaluation stack. Go to step 1.

We illustrate this algorithm in the following diagram, where the expression

`13, 4, -, 2, 3, *, +`

is evaluated.

A two-stack algorithm to evaluate Polish expressions



Let us write a program that implements this two-stack algorithm. A key idea will be to redefine `data` so that it can store either a value in the form of an integer or an operator in the form of a character. Our program will consist of a *.h* file and five *.c* files. Here is our header file:

In file polish.h

```
/* A linked list implementation of a Polish stack. */

#include <assert.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>

#define   EMPTY      0
#define   FULL       10000

struct data {
   enum {operator, value}   kind;
   union {
      char   op;
      int    val;
   }                              u;
};

typedef   struct data             data;
typedef   enum {false, true}      boolean;

struct elem {                      /* an element on the stack */
   data          d;
   struct elem   *next;
};

typedef   struct elem   elem;
```

```
struct stack {
   int    cnt;                    /* a count of the elements */
   elem   *top;                   /* ptr to the top element */
};

typedef   struct stack   stack;
boolean   empty(const stack *stk);
int       evaluate(stack *polish);
void      fill(stack *stk, const char *str);
boolean   full(const stack *stk);
void      initialize(stack *stk);
data      pop(stack *stk);
void      prn_data(data *dp);
void      prn_stack(stack *stk);
void      push(data d, stack *stk);
data      top(stack *stk);
```

Observe that this header file is similar to the one we used in our stack program in Section 10.5, "Stacks," on page 461. The main difference is that we have defined `data` to be a structure type. The structure contains a `union` that can hold either an `int` value or an operator in the form of a `char`. It also contains a "flag" in the form of an enumeration type. The flag will tell us which kind of data is being stored.

In file main.c

```
/* Test the two-stack Polish evaluation algorithm. */

#include "polish.h"

int main(void)
{
   char    str[] = "13, 4, -, 2, 3, *, +";
   stack   polish;

   printf("\n%s%s\n\n",
      "Polish expression: ", str);
   fill(&polish, str);              /* fill stack from string */
   prn_stack(&polish);              /* print the stack */
   printf("\n%s%d\n\n",
      "Polish evaluation: ", evaluate(&polish));
   return 0;
}
```

In `main()`, we fill the Polish stack from a string and print the stack to check that everything is working properly. The function that is of most interest is the one that evaluates the Polish stack. Let us present that function next.

In file eval.c

```
/* Evaluation of the Polish stack. */

#include "polish.h"

int evaluate(stack *polish)
{
   data    d, d1, d2;
   stack   eval;

   initialize(&eval);
   while (!empty(polish)) {
      d = pop(polish);
      switch (d.kind) {
      case value:
         push(d, &eval);
         break;
      case operator:
         d2 = pop(&eval);
         d1 = pop(&eval);
         d.kind = value;          /* begin overwriting d */
       switch (d.u.op) {
       case '+':
          d.u.val = d1.u.val + d2.u.val;
          break;
       case '-':
          d.u.val = d1.u.val - d2.u.val;
          break;
       case '*':
          d.u.val = d1.u.val * d2.u.val;
       }
       push(d, &eval);
      }
   }
   d = pop(&eval);
   return d.u.val;
}
```

The `evaluate()` function embodies the two-stack algorithm that we presented earlier. We first pop `d` from the `polish` stack. If it is a value, we push it onto the `eval` stack. If it is an operator, we pop `d2` and `d1` off the `eval` stack, perform the indicated operation, and push the results onto the `eval` stack. When the `polish` stack is empty, we pop `d` from the `eval` stack and return the `int` value `d.u.val`.

We write the stack operations in the file *stack.c*. Except for the inclusion of a different header file, there is no difference at all from the file *stack.c*, which we wrote in Section 10.5, "Stacks," on page 461.

In file stack.c

```
/* The basic stack routines. */

#include "polish.h"

void initialize(stack *stk)
{
   stk -> cnt = 0;
   stk -> top = NULL;
}

void push(data d, stack *stk)
{
   elem   *p;

   p = malloc(sizeof(elem));
   p -> d = d;
   p -> next = stk -> top;
   stk -> top = p;
   stk -> cnt++;
}
.....
```

In the stack implementation in Section 10.5, "Stacks," on page 461, the type `data` was equivalent to `char`; here `data` is a structure type. By using a `typedef` to embody the idea of "data," we have made the ADT stack implementation reusable. This is an important point. By using code that has already been written and tested, the programmer has less work to do in the current project.

We need to be able to fill a stack from a string that contains a Polish expression. Here is our function that does this:

In file fill.c

```
#include "polish.h"

void fill(stack *stk, const char *str)
{
   const char   *p = str;
   char         c1, c2;
   boolean      b1, b2;
   data         d;
   stack        tmp;

   initialize(stk);
   initialize(&tmp);
```



```
/*
// First process the string and push data on tmp.
*/
while (*p != '\0') {
   while (isspace(*p) || *p == ',')
      ++p;
   b1 = (boolean) ((c1 = *p) == '+' || c1 == '-' || c1 == '*');
   b2 = (boolean) ((c2 = *(p + 1)) == ',' || c2 == '\0');
   if (b1 && b2) {
      d.kind = operator;
      d.u.op = c1;
   }
   else {
      d.kind = value;
      assert(sscanf(p, "%d", &d.u.val) == 1);
   }
   if (!full(&tmp))
      push(d, &tmp);              /* push data on tmp */
   while (*p != ',' && *p != '\0')
      ++p;
}
/*
// Now pop data from tmp and push on stk.
*/
while (!empty(&tmp)) {
   d = pop(&tmp);                 /* pop data from tmp */
   if (!full(stk))
      push(d, stk);               /* push data on stk */
}
}
```

First we process the string to extract the data. As we find the data, we push it onto the stack `tmp`. After we finish processing the string, we pop the data off of `tmp` and push it onto the stack pointed to by `stk`. We do this so that the element in the stack pointed to by `stk` will be in the right order.

We write two printing functions so that we are able to check that our code is working properly. Here are the functions:

In file print.c

```
#include "polish.h"

void prn_data(data *dp)
{
   switch (dp -> kind) {
   case operator:
      printf("%s%3c\n",
       "kind: operator      op:", dp -> u.op);
      break;
   case value:
      printf("%s%3d\n",
       "kind: value        val:", dp -> u.val);
   }
}

void prn_stack(stack *stk)
{
   data   d;

   printf("stack count:%3d%s",
      stk -> cnt, (stk -> cnt == 0) ? "\n" : "     ");
   if (!empty(stk)) {
      d = pop(stk);          /* pop the data */
      prn_data(&d);          /* print the data */
      prn_stack(stk);        /* recursive call */
      push(d, stk);          /* push the data */
   }
}
```

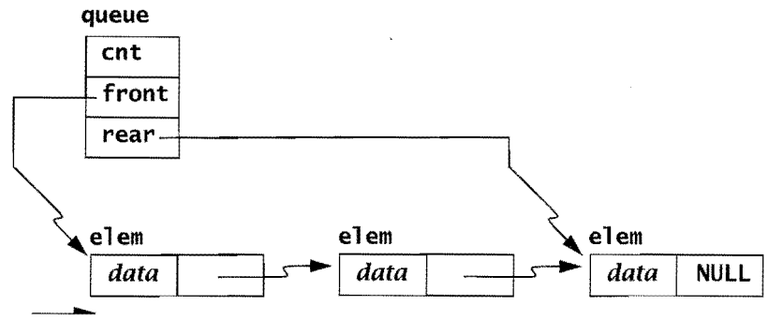
The algorithm to print the stack is quite simple. First, pop d off the stack and print it. Then make a recursive call to `prn_stack()`. Finally, push d back onto the stack. The effect of this is to print all the data in the stack and to leave the stack in its original state. Here is the output from our program:

```
Polish expression: 13, 4, -, 2, 3, *, +

stack count:  7     kind: value        val: 13
stack count:  6     kind: value        val:  4
stack count:  5     kind: operator      op:  -
stack count:  4     kind: value        val:  2
stack count:  3     kind: value        val:  3
stack count:  2     kind: operator      op:  *
stack count:  1     kind: operator      op:  +
stack count:  0

Polish evaluation: 15
```


10.7 Queues

A queue is another abstract data type (ADT) that we can implement as a linear linked list. A queue has a *front* and a *rear*. Insertion occurs at the rear of the list, and deletion occurs at the front of the list.

A queue implementation



Our implementation of the ADT queue will include a number of the standard queue functions. Here is our header file:

In file queue.h

```
/* A linked list implementation of a queue. */

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#define   EMPTY      0
#define   FULL       10000

typedef   unsigned int          data;
typedef   enum {false, true}    boolean;

struct elem {                    /* an element in the queue */
   data          d;
   struct elem   *next;
};
```

```
typedef   struct elem   elem;
struct queue {
   int    cnt;                  /* a count of the elements */
   elem   *front;               /* ptr to the front element */
   elem   *rear;                /* ptr to the rear element */
};

typedef   struct queue   queue;
void      initialize(queue *q);
void      enqueue(data d, queue *q);
data      dequeue(queue *q);
data      front(const queue *q);
boolean   empty(const queue *q);
boolean   full(const queue *q);
```

At the bottom of the header file, we put the list of function prototypes. We write the function definitions in the file *queue.c*. These functions, together with this header file, implement the ADT queue.

In file queue.c

```
/* The basic queue routines. */

#include "queue.h"

void initialize(queue *q)
{
   q -> cnt = 0;
   q -> front = NULL;
   q -> rear = NULL;
}
```

In file queue.c

```
data dequeue(queue *q)
{
   data   d;
   elem   *p;

   d = q -> front -> d;
   p = q -> front;
   q -> front = q -> front -> next;
   q -> cnt--;
   free(p);
   return d;
}
```



```
void enqueue(data d, queue *q)
{
   elem   *p;

   p = malloc(sizeof(elem));
   p -> d = d;
   p -> next = NULL;
   if (!empty(q)) {
      q -> rear -> next = p;
      q -> rear = p;
   }
   else
      q -> front = q -> rear = p;
   q -> cnt++;
}

data front(const queue *q)
{
   return (q -> front -> d);
}

boolean empty(const queue *q)
{
   return ((boolean) (q -> cnt == EMPTY));
}

boolean full(const queue *q)
{
   return ((boolean) (q -> cnt == FULL));
}
```

The `enqueue()` routine uses the storage allocator `malloc()` to create a new queue element, and the `dequeue()` routine returns the freed up storage back to the system.

A queue is a *first-in-first-out* (FIFO) data structure. The first item to be placed onto the queue is the first to be removed. Queues are very useful for a variety of system programming applications. For example, they are often used to schedule resources in an operating system. They are also useful in writing event simulators.

As an example, let us write an elementary resource scheduler for a two-processor system. We will assume that a process can request to be serviced by either processor A or processor B. The process will have a unique process identification (pid) number. After reading all the requests, the scheduler will print the schedule of processes served by each processor.

In file main.c

```
/* Using queues to schedule two resources. */

#include "queue.h"

int main(void)
{
   int     c;
   int     cnt_a = 0;
   int     cnt_b = 0;
   data    pid;              /* process id number */
   queue   a, b;

   initialize(&a);
   initialize(&b);
   /* Enqueue the requests.*/
   while ((c = getchar()) != EOF) {
      switch (c) {
      case 'A':
       assert(scanf("%u", &pid) == 1);
       if (!full(&a))
          enqueue(pid, &a);
       break;
      case 'B':
       assert(scanf("%u", &pid) == 1);
       if (!full(&b))
          enqueue(pid, &b);
      }
    }

    /* Dequeue the requests and print them.*/
    printf("\nA's schedule:\n");
    while (!empty(&a)) {
       pid = dequeue(&a);
       printf("   JOB %u is %d\n", ++cnt_a, pid);
    }
    printf("\nB's schedule:\n");
    while (!empty(&b)) {
       pid = dequeue(&b);
       printf("   JOB %u is %d\n", ++cnt_b, pid);
    }
    return 0;
}
```


To test our program, we create the following input file:

In file input

```
B  7702
A  1023
B  3373
A  5757
A  1007
```

When we give the command

scheduler < input

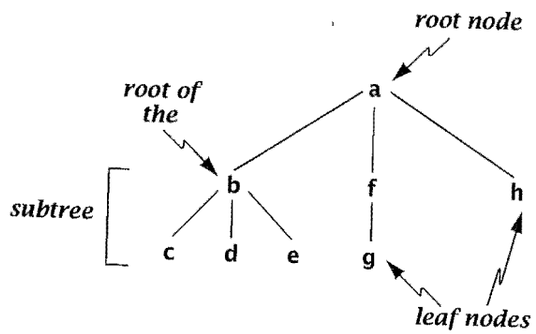
the following gets printed on the screen:

```
A's schedule:
   JOB 1 is 1023
   JOB 2 is 5757
   JOB 3 is 1007
B's schedule:
   JOB 1 is 7702
   JOB 2 is 3373
```

10.8 Binary Trees

A *tree* is a finite set of elements called *nodes*. A tree has a unique node, called the *root* node, where the remaining nodes are a disjoint collection of subtrees. If node r has $T_1, T_2, \ldots, T_n$ as subtrees, then $r_1, r_2, \ldots, r_n$, the roots of these subtrees, are the children of r. A node with no children is called a *leaf* node.

A general tree



A *binary tree* is a tree whose elements have two children. A binary tree considered as a data structure is an object made up of elements that are characterized by two link fields, called left child and right child. Each link must point at a new object not already pointed at or be NULL. In this representation, a leaf node has both left and right child as the value NULL. The following structures and type specifications will be used to define binary trees.

In file tree.h

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

typedef   char   DATA;

struct node {
   DATA          d;
   struct node   *left;
   struct node   *right;
};

typedef   struct node   NODE;
typedef   NODE          *BTREE;

#include "fct_proto.h"              /* function prototypes */
```

The file *tree.h* must be included with all binary tree functions defined in this section.

A key advantage of a binary tree over a linear list is that elements are normally reached, on average, in a logarithmic number of link traversals. This gain in time efficiency for retrieving information is at the expense of the space needed to store the extra link field per element. We illustrate this advantage in the exercises.

Binary Tree Traversal

There is one way to march down a linear list, namely from head to tail. However, there are several natural ways to visit the elements of a binary tree. The three commonest are

Inorder:	left subtree	*Preorder:*	root	*Postorder:*	left subtree
	root		left subtree		right subtree
	right subtree		right subtree		root

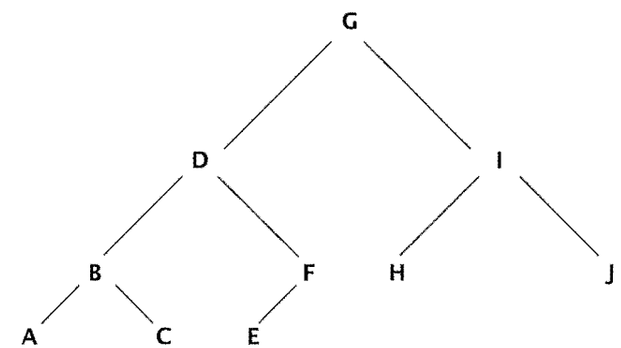
These standard methods of visitation are the basis for recursive algorithms that manipulate binary trees.

```
/* Inorder binary tree traversal. */

void inorder(BTREE root)
{
   if (root != NULL) {
      inorder(root -> left);       /* recur left */
      printf("%c ", root -> d);
      inorder(root -> right);      /* recur right */
   }
}
```

The function inorder() will print the values of each node in the binary tree pointed at by root. The pictured binary tree would be traversed by inorder(), printing

```
A  B  C  D  E  F  G  H  I  J
```

A binary tree



The corresponding preorder and postorder functions are as follows:

```
/* Preorder and postorder binary tree traversal. */

void preorder(BTREE root)
{
   if (root != NULL) {
      printf("%c ", root -> d);
      preorder(root -> left);
      preorder(root -> right);
   }
}

void postorder(BTREE root)
{
   if (root != NULL) {
      postorder(root -> left);
      postorder(root -> right);
      printf("%c ", root -> d);
   }
}
```

Preorder visitation of the binary tree just shown would print

```
G  D  B  A  C  F  E  I  H  J
```

Postorder visitation would print

```
A  C  B  E  F  D  H  J  I  G
```

The reader unfamiliar with these methods should carefully verify these results by hand. Visitation is at the heart of most tree algorithms.

Creating Trees

We will create a binary tree from data values stored as an array. As with lists, we will use the dynamic storage allocator `malloc()`.

```
/* Creating a binary tree. */

BTREE new_node(void)
{
   return (malloc(sizeof(NODE)));
}
```



```
BTREE init_node(DATA d1, BTREE p1, BTREE p2)
{
   BTREE   t;

   t = new_node();
   t -> d = d1;
   t -> left = p1;
   t -> right = p2;
   return t;
}
```

We will use these routines as primitives to create a binary tree from data values stored in an array. There is a very nice mapping from the indices of a linear array into nodes of a binary tree. We do this by taking the value `a[i]` and letting it have as child values `a[2*i+1]` and `a[2*i+2]`. Then we map `a[0]` into the unique root node of the resulting binary tree. Its left child will be `a[1]`, and its right child will be `a[2]`. The function `create_tree()` embodies this mapping. The formal parameter `size` is the number of nodes in the binary tree.

```
/* Create a linked binary tree from an array. */

BTREE create_tree(DATA a[], int i, int size)
{
   if (i >= size)
      return NULL;
   else
      return (init_node(a[i],
              create_tree(a, 2 * i + 1, size),
              create_tree(a, 2 * i + 2, size)));
}
```

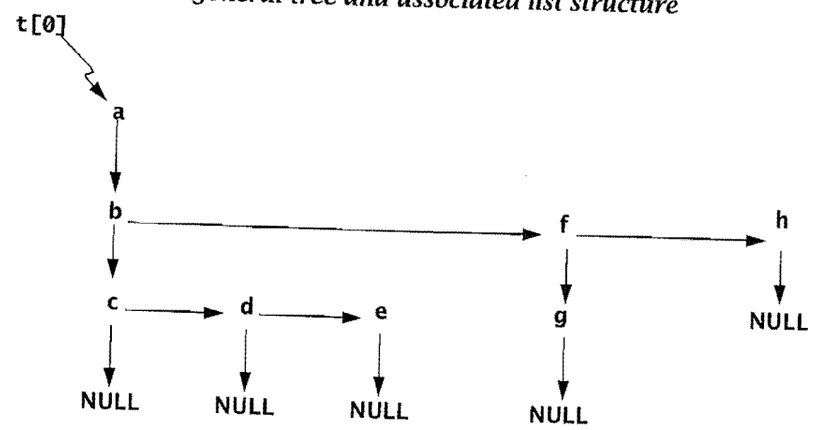
10.9 General Linked Lists

For some data structures, we wish to combine the use of arrays with the use of lists. The arrays provide random accessing, and the lists provide sequential accessing. We will show one such example in this section, an implementation of a general tree. In a general tree, a node can have an arbitrary number of children. It would be very wasteful to specify a structure using the maximum number of links for each node.

We will represent a general tree as a number of linear linked lists, one for each node in the tree. Each list will be the children of a single node. The lists will have an array that will point at the first child of the corresponding node. The base element of the

array will point at the root node. The following diagram shows such a representation, in this case for the general tree at the beginning of Section 10.8, "Binary Trees," on page 476:

A general tree and associated list structure



Such trees can be represented using the following header file:

In file gtree.h

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

typedef   char   DATA;

struct node {
   int           child_no;
   DATA          d;
   struct node   *sib;
};

typedef   struct node   NODE;
typedef   NODE          *GTREE;

#include "fct_proto.h"              /* function prototypes */
```


We will use an array of type GTREE, say t, where t[0] points to the root element represented as type NODE. Siblings will be reached by linear list chaining, and children by array indexing. Let us examine the creation of a particular such tree. We first write routines to create a single node.

```
/* Create a new node. */

GTREE new_gnode(void)
{
   return (malloc(sizeof(NODE)));
}

GTREE init_gnode(DATA d1, int num, GTREE sibs)
{
   GTREE   tmp;

   tmp = new_gnode();
   tmp -> d = d1;
   tmp -> child_no = num;
   tmp -> sib = sibs;
   return tmp;
}
```

Let us use these routines to create the tree of the previous diagram. Because it contains eight nodes, we need an array t[] of size 9 and type NODE, where t[0] is the pointer to the root node.

```
t[0] = init_gnode('a', 1, NULL);
t[1] = init_gnode('b', 2, NULL);
t[1] -> sib = init_gnode('f', 6, NULL);
t[1] -> sib -> sib = init_gnode('h', 8, NULL);
t[2] = init_gnode('c', 3, NULL);
t[2] -> sib = init_gnode('d', 4, NULL);
t[2] -> sib -> sib = init_gnode('e', 5, NULL);
t[3] = NULL;
t[4] = NULL;
t[5] = NULL;
t[6] = init_gnode('g', 7, NULL);
t[7] = NULL;
t[8] = NULL;
```

It is easy to detect certain properties in this representation, such as whether a node is a leaf node or how many children a node has. For node n, if t[n] points at NULL, then the node is a leaf.

Traversal

Traversal becomes a combination of (1) moving along lists and (2) indexing into array elements that point at the lists. Generalizing the traversal ordering of preorder, postorder, and inorder to these structures is quite straightforward. Once again, these algorithms are prototypes for more complicated functions that can be done on a tree, because they guarantee that each element will be reached in linear time.

```
/* Preorder traversal of general trees. */

void preorder_g(GTREE t, int ind)
{
   GTREE   tmp;    /* tmp traverses the sibling list */

   tmp = t[ind];              /* t[ind] is the root node */
   while (tmp != NULL) {
      printf("%c  %d\n", tmp -> d, tmp -> child_no);
      preorder_g(t, tmp -> child_no);
      tmp = tmp -> sib;
   }
}
```

The function `preorder_g()` differs from the corresponding binary tree function in that a `while` loop is necessary to move along the linear list of siblings. Notice that recursion allows each subtree to be handled cleanly.

The Use of `calloc()` and Building Trees

The library function `calloc()` provides contiguous allocation of storage that can be used for arrays. Its function prototype is given in *stdlib.h* as

```
void *calloc(size_t n, size_t size);
```

Thus, the arguments to `calloc()` are converted to type `size_t`, and the value returned is of type pointer to `void`. Typically, `size_t` is equivalent to `unsigned`. A function call of the form

`calloc(`*n*, *size*`)`

returns a pointer to enough contiguous storage for *n* objects, each of *size* bytes. This storage is initialized to zero by the system. Thus, `calloc()` can be used to dynamically allocate storage for a run-time defined array. This allows the user to allocate space as

needed rather than specifying during compilation an array size that is very large so as to accommodate all cases that might be of interest.

For example, we will want a routine that can build a general tree from a list of edges and an array of type DATA. If we wish to have an array of size 10 to store the tree headers, we can write

```
t = calloc(10, sizeof(GTREE));
```

Now we can pass the dynamically allocated array `t` of type pointer to GTREE to a function `buildtree()` in order to construct a general tree. This function will take an edge list representation of a tree and compute its general list structure representation.

```
/* Function buildtree creates a tree from an array of edges. */

typedef struct {     /* PAIR represents an edge in a tree */
   int   out;
   int   in;
} PAIR;

void buildtree(PAIR edges[], DATA d[], int n, GTREE t[])
{
   int   i;
   int   x, y;                          /* points of edge */

   t[0] = init_gnode(d[1], 1, NULL);
                                  /* t[0] takes node 1 as root */
   for (i = 1; i <= n; ++i)
      t[i] = NULL;
   for (i = 0; i < n - 1; ++i) {
      x = edges[i].out;
      y = edges[i].in;
      t[x] = init_gnode(d[y], y, t[x]);
   }
}
```

Similar data structures and functions can be used to develop representations of general graphs, sparse matrices, and complicated networks.

Summary

1 Self-referential structures use pointers to address identically specified elements.

2 The simplest self-referential structure is the linear linked list. Each element points to its next element, with the last element pointing at NULL, defined as 0.

3 The function `malloc()` is used to dynamically allocate storage. It takes an argument of type `size_t` and returns a pointer to `void` that is the address of the allocated storage.

4 The function `free()` is a storage management routine that returns to available storage the block of memory pointed at by its argument.

5 Standard algorithms for list processing are naturally implemented recursively. Frequently, the base case is the detection of the NULL link. The general case recurs by moving one link over in the list structure.

6 When algorithms are implemented iteratively, one uses an iterative loop, terminating when NULL is detected. Iterative algorithms trade the use of auxiliary pointers for recursion.

7 The abstract data type (ADT) stack is implementable as a linked list, with access restricted to its first element, which is called the top. The stack has a LIFO (last-in-first-out) discipline implemented by the routines `push()` and `pop()`.

8 The ADT queue is also implementable as a linked list, with access restricted to its front and rear ends. The queue has a FIFO (first-in-first-out) discipline implemented by the routines `enqueue()` and `dequeue()`.

9 Binary trees are represented as structures with two link members and combine the dynamic qualities of linear lists with, on average, significantly shorter access paths to each element. The distances to elements of binary trees are usually logarithmic.

10 Binary trees are traversed most often in one of three major patterns: preorder, inorder, or postorder. Each ordering is determined by when the root is visited. Preorder visits the root first; inorder, after the left subtree; and postorder, last. These traversal patterns are readily implemented as recursions, chaining down both left and right subtrees.

11 Data structures of formidable complexity, involving both lists and arrays, can be specified. One example of their use is in the implementation of a general tree, where a node has an arbitrary number of children. The children of a node are represented as a list pointed to by an array of header elements.

Exercises

1 Try the following code. Why does your compiler complain? Fix the code so it compiles correctly:

```
struct husband {
   int           age;
   char          name[10];
   struct wife   spouse;
} a;

struct wife {
   int              age;
   char             name[10];
   struct husband   spouse;
} b;
```

2 Change the type definition of DATA in the file *list.h* to

```
typedef struct {
   char   name[10];
   int    age;
   int    weight;
}  DATA;
```

and write a function `create_l()` that transforms an array of such data into a linear list. Write another routine, one that will count the number of people above a given weight and age.

3 Given a linear list of the type found in the previous exercise, write a routine `sort_age()` that will sort the list according to its age values. Write a function `sort_name()` that will sort in lexicographic order based on the name values.

4 Combine the sorting functions in the previous exercise so that they share the most code. This is best done by defining a routine called `compare()` that returns either 0

or 1, depending on which element is larger. Use this function as a parameter to a linear list sorting function.

5 Draw the list that would result from `concatenate(a, a)`, where a points at a list of two elements. What happens if the resulting list pointed at by a is passed to `print_list(a)`?

6 The previous exercise was used to construct a cycle. A cycle is a pointer chain that points back to itself. Cycles are particularly nasty run-time bugs that can be hard to recognize. Write a function `iscycle(head)` that returns 1 if a cycle is detected and 0 otherwise. *Hint:* Save the address of the initial element of the list and move around until either NULL is reached or the initial address is encountered.

7 Write an iterative version of the function `print_list()`.

8 Modify `concatenate()` so that it returns the address of the head of the resulting list. Also, if the first list is NULL, it should return the second list. Have it test to see if both lists are the same. If they are, a cycle will result, and the program should return a warning to that effect. (See exercises 5 and 6 above.)

9 Write a routine `copy_cat(a, b)` that returns a concatenated copy of the lists a and b. The original lists a and b should remain undisturbed.

10 Write an iterative version of the function `concatenate()` that you wrote in exercise 8 above.

11 Write an insertion function that inserts an element at the head of the list.

12 Write an insertion function that inserts an element at the tail of the list.

13 Write an insertion function that inserts an element at the first position in the list following an element storing a particular DATA item.

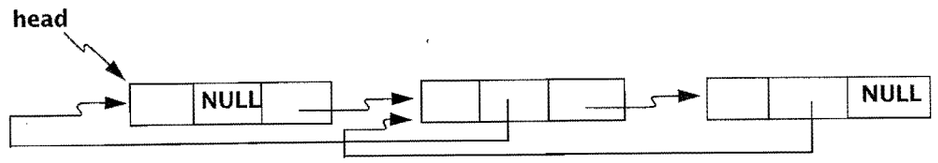
14 Generalize the previous three exercises. Write an insertion function that inserts an element in the *n*th position in a list, where 0 means the element is placed at the head of the list. If *n* is larger than the length of the list, insert the element at the tail of the list.

15 An element of a doubly linked linear list can be defined as

```
typedef struct dllist {
   DATA            d;
   struct dllist   *prev;
   struct dllist   *next;
} ELEMENT;
```

This adds an extra member but allows easier traversal along the list.

A doubly linked list



Write iterative routines to perform insertion and deletion.

16 Write a routine `del_dupl()` that deletes duplicate valued elements in a doubly linked list.

17 Evaluate the following Polish expressions by hand:

```
7, 6, -, 3, *
9, 2, 3, *, 4, -, +
1, 2, +, 3, 4, +, *
```

18 Write corresponding Polish expressions for the following:

```
(7 + 8 + 9) * 4
(6 - 2) * (5 + 15 * 2)
6 - 2 * 5 + 15 * 2
```

19 Use the *polish* program that we wrote in Section 10.6, "An Example: Polish Notation and Stack Evaluation," on page 466, to evaluate the six Polish expressions given or derived from the two above exercises.

20 The following lines occur in the `evaluate()` function in the *polish* program:

```
case operator:
   d2 = pop(&eval);
   d1 = pop(&eval);
```

What happens if we write

```
case operator:
   d1 = pop(&eval);
   d2 = pop(&eval);
```

instead? The program should work properly on some Polish expressions but not on others. Explain.

21 Create another version of the *polish* program. First, write a routine that reverses the order of the elements in a stack. Next, append this routine to the file *stack.c*, and add the function prototype to the header file. Then rewrite the function `fill()` so that it makes use of your routine. Finally, test your program.

22 Write a routine that allows you to interactively initialize the contents of a Polish stack. Write a program to test your routine.

23 When we tested our *polish* program, the Polish expression we used did not have any unary operators in it. Try the program with the following expression:

```
2, -3, -, -4, 7, +, -1, -1, +, *, *
```

Is the *polish* program able to handle this correctly? (It should be able to.) Write another version of the *polish* program that can handle both unary plus and unary minus operators in the list. For example, your program should be able to handle the following Polish expression:

```
2, -, 3, -, -, 4, +, 7, +, -1, -, +, +, 1, +, *, *
```

In this expression, some of the plus and minus signs are unary and some are binary.

24 Write a function whose prototype is

```
queue data_to_queue(data d[], int n);
```

where `n` is the size of the array `d`. When passed an array of type `data`, this function should use the elements in the array to build a queue that gets returned.

25 Write a function whose prototype is

```
data *queue_to_data(queue q);
```

that builds a `data` array out of the values found in the queue. The size of the array should be `q.cnt`.

26 When we tested the scheduler program that we wrote in Section 10.7, "Queues," on page 474, our input file was rather simple. Try the following input:

```
x x x A 2323  B 1188  yyy zzz C 3397 3398 A 4545  X  surprise?
```

Are you surprised that it works? Explain.

27 Write a four-resource version of the scheduler program.

28 Modify the program that you wrote for the previous exercise to allow for priorities. Change the type `data` to a structure that has both a processor ID number and a priority. The priority should be an unsigned number between 0 and 9. Create schedules where the highest priority processes are serviced first. *Caution:* Do not confuse the concepts of "process" and "processor."

29 Write routines for binary trees that

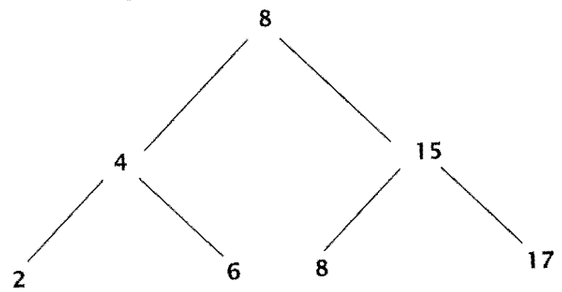
1 count the number of nodes

2 count the number of nodes having a particular value, say 'b'

3 print out the value of only leaf nodes

30 Create a binary tree from an array of `int`s such that a left child has value less than its root's value, and a right child has value greater than or equal to the value of its root. Such a tree is displayed in the following diagram:

A binary tree with ordered values

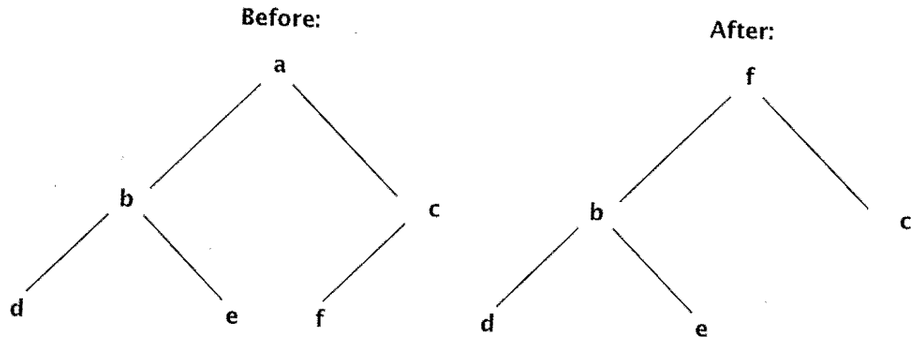


Insert a new element by comparing its value to the root and recurring down the proper subtree until NULL is reached.

31 For the tree of the previous exercise, write a function that uses inorder traversal to place the values of the nodes in sorted order in an array key[].

32 Write a program that deletes the root node of a binary tree and replaces the root with the rightmost leaf node.

Deleting the root node with replacement



33 (Advanced) Write heapsort(). This is a sorting routine based on an ordered binary tree. A heap is a binary tree in which a node has value smaller than any of its children. This property guarantees that the root is the smallest element in the tree. We wish to take an unordered array of values and create a heap with these values as data entries.

Given a heap

1 Delete the root, placing it in an output array as the smallest element.

2 Take the rightmost leaf node and make it the new root.

3 Compare the root value to both children and exchange with the smaller of the two values.

4 Continue to exchange values until the current node is smaller in value than its children.

Now the tree is once again a heap. Repeat these steps until the tree is empty and the array has the original tree values in sorted order. You must figure out how to get

the tree into heap order to begin with; see Aho, Hopcroft, and Ullman, *Data Structures and Algorithms* (Reading, MA: Addison-Wesley, 1987).

34 Print out the nodes of a binary tree in level order. The root node is at level 0. On the next level of nodes are the children of the previous level. First, print the root node. Then print the left child and the right child that are at level 1. Continue printing the nodes from left to right, level by level. This is also called breadth-first traversal.

35 Write a routine that computes the maximum level leaf node of a general tree.

36 Write a function that converts a binary tree into a general tree representation.

37 Add a field called weight to our general tree structure. Write functions to

1 compute the sum of all node weights

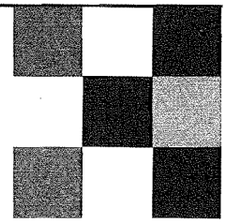
2 compute the maximum weighted path, where the weighted path of node *i* is the weights of the nodes from the root to node *i*

It can be proved that the maximum weighted path occurs at a leaf, given that all weights are nonnegative.

38 Use a general linked list structure to program sparse matrix addition. A sparse matrix is one in which most of its values are zero. A nonzero element of a sparse matrix will be represented by the triple (*i*, *j*, *value*). For each row *i*, the triples will be linked as a linear list headed by an array of pointers row[i]. Similarly, for each column *j*, the triples will be linked as a linear list headed by an array of pointers col[j]. To add matrix *A* to matrix *B*, we take each row and merge them into a matrix *C*. If both rows have a triple with the same column value, then in the output row $c_{i,j}$ is the sum $a_{i,j} + b_{i,j}$. Otherwise, the element with smallest column number becomes the next element of the output row.

39 (Advanced) The representation in the previous exercise can also be used to do sparse matrix multiplication. Although addition can be performed with just row-linked lists, both row- and column-linked lists are used for multiplication. Write a program to do sparse matrix multiplication.

40 (Project) Implement the sparse polynomial ADT. Use a linked list of elements to represent the nonzero terms of the polynomial. A term is a real coefficient and a power. Write a complete polynomial manipulation package. Your package should be able to input and output polynomials, and it should be able to add, subtract, multiply, and copy polynomials.



Chapter 11

Input/Output and the Operating System

In this chapter, we explain how to use some of the input/output functions in the standard library, including the functions `printf()` and `scanf()`. Although we have used these functions throughout this text, many details still need to be explained. We present extensive tables showing the effects of various formats. The standard library provides functions related to `printf()` and `scanf()` that can be used for dealing with files and strings. The use of these functions is explained.

General file input/output is important in applications where data reside in files on disks and tapes. We will show how to open files for processing and how to use a pointer to a file. Some applications need temporary files. Examples are given to illustrate their use.

The operating system provides utilities for the user, and some of these utilities can be used by the programmer to develop C software. A number of the more important tools for programmers will be discussed, including the compiler, *make*, *touch*, *grep*, beautifiers, and debuggers.

11.1 The Output Function `printf()`

The `printf()` function has two nice properties that allow flexible use at a high level. First, a list of arguments of arbitrary length can be printed, and second, the printing is controlled by simple conversion specifications, or formats. The function `printf()` delivers its character stream to the standard output file `stdout`, which is normally connected to the screen. The argument list to `printf()` has two parts:

control_string and *other_arguments*

In the example

```
printf("she sells %d %s for f", 99, "sea shells", 3.77);
```

we have

control_string: `"she sells %d %s for f"`
other_arguments: `99, "sea shells", 3.77`

The expressions in *other_arguments* are evaluated and converted according to the formats in the control string and then placed in the output stream. Characters in the control string that are not part of a format are placed directly in the output stream. The % symbol introduces a conversion specification, or format. A single-conversion specification is a string that begins with % and ends with a conversion character:

printf() conversion characters

Conversion character	How the corresponding argument is printed
c	as a character
d, i	as a decimal integer
u	as an unsigned decimal integer
o	as an unsigned octal integer
x, X	as an unsigned hexadecimal integer
e	as a floating-point number; example: 7.123000e+00
E	as a floating-point number; example: 7.123000E+00
f	as a floating-point number; example: 7.123000
g	in the e-format or f-format, whichever is shorter
G	in the E-format or f-format, whichever is shorter
s	as a string
p	the corresponding argument is a pointer to void; its value is printed as a hexadecimal number
n	the corresponding argument is a pointer to an integer into which the number of characters written so far is printed; the argument is not converted
%	with the format %% a single % is written to the output stream; there is no corresponding argument to be converted

The function `printf()` returns as an `int` the number of characters printed. In the example

```
printf("she sells %d %s for f", 99, "sea shells", 3.77);
```

we can match the formats in the control string with their corresponding arguments in the argument list.

Format	Corresponding argument
%d	99
%s	"sea shells"
%f	3.77

Explicit formatting information may be included in a conversion specification. If it is not included, then defaults are used. For example, the format `%f` with corresponding argument `3.77` will result in `3.770000` being printed. The number is printed with six digits to the right of the decimal point by default.

Between the % that starts a conversion specification and the conversion character that ends it, there may appear in order

- zero or more flag characters that modify the meaning of the conversion specification. These flag characters are discussed below.

- an optional positive integer that specifies the minimum field width of the converted argument. The place where an argument is printed is called its field, and the number of spaces used to print an argument is called its field width. If the converted argument has fewer characters than the specified field width, then it will be padded with spaces on the left or right, depending on whether the converted argument is right- or left-adjusted. If the converted argument has more characters than the specified field width, then the field width will be extended to whatever is required. If the integer defining the field width begins with a zero and the argument being printed is right-adjusted in its field, then zeros rather than spaces will be used for padding.

- an optional *precision*, which is specified by a period followed by a nonnegative integer. For `d`, `i`, `o`, `u`, `x`, or `X` conversions, it specifies the minimum number of digits to be printed. For `e`, `E`, and `f` conversions, it specifies the number of digits to the right of the decimal point. For `g` and `G` conversions, it specifies the maximum number of significant digits. For an `s` conversion, it specifies the maximum number of characters to be printed from a string.

- an optional h or l, which is a "short" or "long" modifier, respectively. If an h is followed by a d, i, o, u, x, or X conversion character, the conversion specification applies to a short int or unsigned short int argument. If an h is followed by an n conversion character, the corresponding argument is a pointer to a short int or unsigned short int. If an l is followed by a d, i, o, u, x, or X conversion character, the conversion specification applies to a long int or unsigned long int argument. If an l is followed by an n conversion character, the corresponding argument is a pointer to a long int or unsigned long int.

- an optional L, which is a "long" modifier. If an L is followed by an e, E, f, g, or G conversion character, the conversion specification applies to a long double argument.

The flag characters are

- a minus sign, which means that the converted argument is to be left-adjusted in its field. If there is no minus sign, then the converted argument is to be right-adjusted in its field.

- a plus sign, which means that a nonnegative number that comes from a signed conversion is to have a + prepended. This works with the conversion characters d, i, e, E, f, g, and G. All negative numbers start with a minus sign.

- a space, which means that a nonnegative number that comes from a signed conversion is to have a space prepended. This works with the conversion characters d, i, e, E, f, g, and G. If both a space and a + flag are present, the space flag is ignored.

- a #, which means that the result is to be converted to an "alternate form" that depends on the conversion character. With conversion character o, the # causes a zero to be prepended to the octal number being printed. In an x or X conversion, the # causes 0x or 0X to be prepended to the hexadecimal number being printed. In a g or G conversion, it causes trailing zeros to be printed. In an e, E, f, g, and G conversion, it causes a decimal point to be printed, even with precision 0. The behavior is undefined for other conversions.

- a zero, which means that zeros instead of spaces are used to pad the field. With d, i, o, u, x, X, e, E, f, g, and G conversion characters, this can result in numbers with leading zeros. Any sign and any 0x or 0X that gets printed with a number will precede the leading zeros.

In a format, the field width or precision or both may be specified by a * instead of an integer, which indicates that a value is to be obtained from the argument list. Here is an example of how the facility can be used:


```
int      m, n;
double   x = 333.7777777;
.....                                  /* get m and n from somewhere */
printf("x = %*.*f\n", m, n, x);
```

If the argument corresponding to the field width has a negative value, then it is taken as a – flag followed by a positive field width. If the argument corresponding to the precision has a negative value, then it is taken as if it were missing.

The conversion specification %% can be used to print a single percent symbol in the output stream. It is a special case because there is no corresponding argument to be converted. For all the other formats, there should be a corresponding argument. If there are not enough arguments, the behavior is undefined. If there are too many arguments, the extra ones are evaluated but otherwise ignored.

The field width is the number of spaces used to print the argument. The default is whatever is required to properly display the value of the argument. Thus, the integer value 255 (decimal) requires three spaces for decimal conversion d or octal conversion o, but only two spaces for hexadecimal conversion x.

When an argument is printed, characters appropriate to the conversion specification are placed in a field. The characters appear right-adjusted unless a minus sign is present as a flag. If the specified field width is too short to properly display the value of the corresponding argument, the field width will be increased to the default. If the entire field is not needed to display the converted argument, then the remaining part of the field is padded with blanks on the left or right, depending on whether the converted argument is right- or left-adjusted. The padding character on the left can be made a zero by specifying the field width with a leading zero.

The precision is specified by a nonnegative number that occurs to the right of the period. For string conversions, this is the maximum number of characters to be printed from the string. For e, E, and f conversions, it specifies the number of digits to appear to the right of the decimal point.

Examples of character and string formats are given in the next table. We use double-quote characters to visually delimit the field. They do not get printed.

Declarations and initializations			
`char c = 'A', s[] = "Blue moon!";`			
Format	Corresponding argument	How it is printed in its field	Remarks
`%c`	`c`	`"A"`	field width 1 by default
`%2c`	`c`	`" A"`	field width 2, right adjusted
`%-3c`	`c`	`"A "`	field width 3, left adjusted
`%s`	`s`	`"Blue moon!"`	field width 10 by default
`%3s`	`s`	`"Blue moon!"`	more space needed
`%.6s`	`s`	`"Blue m"`	precision 6
`%-11.8s`	`s`	`"Blue moo "`	precision 8, left adjusted

Examples of formats used to print numbers are given in the next table. Again, we use double-quote characters to visually delimit the field. They do not get printed.

Declarations and initializations			
`int i = 123;` `double x = 0.123456789;`			
Format	Corresponding argument	How it is printed in its field	Remarks
`%d`	`i`	`"123"`	field width 3 by default
`%05d`	`i`	`"00123"`	padded with zeros
`%7o`	`i`	`"  173"`	right adjusted, octal
`%-9x`	`i`	`"7b  "`	left adjusted, hexadecimal
`%-#9x`	`i`	`"0x7b  "`	left adjusted, hexadecimal
`%10.5f`	`x`	`"  0.12346"`	field width 10, precision 5
`%-12.5e`	`x`	`"1.23457e-01 "`	left adjusted, e-format

11.2 The Input Function scanf()

The function `scanf()` has two nice properties that allow flexible use at a high level. The first is that a list of arguments of arbitrary length can be scanned, and the second is that the input is controlled by simple conversion specifications, or formats. The function `scanf()` reads characters from the standard input file `stdin`. The argument list to `scanf()` has two parts:

control_string and *other_arguments*

In the example

```
char    a, b, c, s[100];
int     n;
double  x;

scanf("%c%c%c%d%s%lf", &a, &b, &c, &n, s, &x);
```

we have

control_string: `"%c%c%c%d%s%lf"`
other_arguments: `&a, &b, &c, &n, s, &x`

The other arguments following the control string consist of a comma-separated list of pointer expressions, or addresses. Note that in the preceding example, writing `&s` would be wrong; the expression `s` by itself is an address.

scanf() conversion characters		
Conversion character	Characters in the input stream that are matched	Corresponding argument is a pointer to
c	any character, including white space	char
d, i	an optionally signed decimal integer	integer
u	an optionally signed decimal integer	unsigned integer
o	an optionally signed octal integer	unsigned integer
x, X	an optionally signed hexadecimal integer	unsigned integer
e, E, f, g, G	an optionally signed floating-point number	a floating type
s	a sequence of nonwhite space characters	char
p	what is produced by %p in printf(), usually an unsigned hexadecimal integer	void *
n, %, [...]	see the next table	

Three conversion characters are of a special nature, and one of these, [...], is not even a character, although the construct is treated as such.

scanf() conversion characters	
Conversion character	Remarks
n	No characters in the input stream are matched. The corresponding argument is a pointer to an integer, into which gets stored the number of characters read so far.
%	A single % character in the input stream is matched. There is no corresponding argument.
[...]	The set of characters inside the brackets [ ] is called the scan set. It determines what gets matched. (See the following explanation.) The corresponding argument is a pointer to the base of an array of characters that is large enough to hold the characters that are matched, including a terminating null character that is appended automatically.

The control string may contain

- white space, which matches optional white space in the input stream.
- ordinary nonwhite space characters, other than %. Each ordinary character must match the next character in the input stream.
- conversion specifications that begin with a % and end with a conversion character. Between the % and the conversion character, there may be an optional * that indicates assignment suppression, followed by an optional integer that defines a maximum *scan* width, followed by an optional h, l, or L that modifies the specification character.
- the modifier h, which can precede a d, i, o, u, x, X conversion character. It indicates that the converted value is to be stored in a short int or in an unsigned short int.
- the modifier l, which can precede either a d, i, o, u, x, X conversion character or an e, E, f, g, G conversion character. In the first case, it indicates that the converted value is to be stored in a long int or in an unsigned long int. In the second case, it indicates that the converted value is to be stored in a double.
- the modifier L, which can precede an e, E, f, g, G conversion character. It indicates that the converted value is to be stored in a long double.

The characters in the input stream are converted to values according to the conversion specifications in the control string and placed at the address given by the corresponding pointer expression in the argument list. Except for character input, a scan field consists of contiguous nonwhite characters that are appropriate to the specified conversion. The scan field ends when a nonappropriate character is reached, or the scan width, if specified, is exhausted, whichever comes first. When a string is read in, it is presumed that enough space has been allocated in memory to hold the string and an end-of-string sentinel \0, which will be appended. The format %1s can be used to read in the next nonwhite character. It should be stored in a character array of size at least 2. The format %*n*c can be used to read in the next *n* characters, including white space characters. When one or more characters are read in, white space is not skipped. As with strings, it is the programmer's responsibility to allocate enough space to store these characters. In this case a null character is not appended. A format such as %lf can be used to read in a double. Floating numbers in the input stream are formatted as an optional sign followed by a digit string with an optional decimal point, followed by an optional exponent part. The exponential part consists of e or E, followed by an optional sign followed by a digit string.

A conversion specification of the form %[*string*] indicates that a special string is to be read in. If the first character in *string* is not a circumflex character ^, then the string is to be made up only of the characters in *string*. However, if the first character in *string* is a circumflex, then the string is to be made up of all characters other than those in *string*. Thus, the format %[abc] will input a string containing only the letters *a*, *b*, and *c*, and will stop if any other character appears in the input stream, including a blank. The format %[^abc] will input a string terminated by any of *a*, *b*, or *c*, but not by white space. The statement

```
scanf("%[AB \n\t]", s);
```

will read into the character array s a string containing A's, B's, and the white space characters blank, newline, and tab.

These conversion specifications interact in a predictable way. The scan width is the number of characters scanned to retrieve the argument value. The default is whatever is in the input stream. The specification %s skips white space and then reads in nonwhite space characters until a white space character is encountered or the end-of-file mark is encountered, whichever comes first. In contrast to this, the specification %5s skips white space and then reads in nonwhite characters, stopping when a white space character is encountered or an end-of-file mark is encountered or five characters have been read in, whichever comes first.

The function scanf() returns the number of successful conversions performed. The value EOF is returned when the end-of-file mark is reached. Typically, this value is -1. The value 0 is returned when no successful conversions are performed, and this value is always different from EOF. An inappropriate character in the input stream can frustrate expected conversions, causing the value 0 to be returned. As long as the input stream can be matched to the control string, the input stream is scanned and values are converted and assigned. The process stops if the input is inappropriate for the next conversion specification. The value returned by scanf() can be used to test that input occurred as expected, or to test that the end of the file was reached.

An example illustrating the use of scanf() is

```
int    i;
char   c;
char   string[15];

scanf("%d , %*s %% %c %5s %s", &i, &c, string, &string[5]);
```

With the following characters in the input stream

```
45 , ignore_this  %  C  read_in_this**
```


the value 45 is placed in i, the comma is matched, the string "ignore_this" is ignored, the % is matched, the character C is placed in the variable c, the string "read_" is placed in string[0] through string[5] with the terminating \0 in string[5], and finally the string "in_this**" is placed in string[5] through string[14], with string[14] containing \0. Because four conversions were successfully made, the value 4 is returned by scanf().

11.3 The Functions fprintf(), fscanf(), sprintf(), and sscanf()

The functions fprintf() and fscanf() are file versions of the functions printf() and scanf(), respectively. Before we discuss their use, we need to know how C deals with files.

The identifier FILE is defined in *stdio.h* as a particular structure, with members that describe the current state of a file. To use files, a programmer need not know any details concerning this structure. Also defined in *stdio.h* are the three file pointers stdin, stdout, and stderr. Even though they are pointers, we sometimes refer to them as files.

Written in C	Name	Remark
stdin	standard input file	connected to the keyboard
stdout	standard output file	connected to the screen
stderr	standard error file	connected to the screen

The function prototypes for file handling functions are given in *stdio.h*. Here are the prototypes for fprintf() and fscanf():

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

A statement of the form

```
fprintf(file_ptr, control_string, other_arguments);
```

writes to the file pointed to by *file_ptr*. The conventions for *control_string* and *other_arguments* conform to those of printf(). In particular,

`fprintf(stdout, ...);` is equivalent to `printf(...);`

In a similar fashion, a statement of the form

`fscanf(`*file_ptr*`,` *control_string*`,` *other_arguments*`);`

reads from the file pointed to by *file_ptr*. In particular,

`fscanf(stdin, ...);` is equivalent to `scanf(...);`

In the next section, we will show how to use `fopen()` to open files and how to use `fprintf()` and `fscanf()` to access them.

The functions `sprintf()` and `sscanf()` are string versions of the functions `printf()` and `scanf()`, respectively. Their function prototypes, found in *stdio.h*, are

```
int   sprintf(char *s, const char *format, ...);
int   sscanf(const char *s, const char *format, ...);
```

The function `sprintf()` writes to its first argument, a pointer to `char` (string), instead of to the screen. Its remaining arguments conform to those for `printf()`. The function `sscanf()` reads from its first argument instead of from the keyboard. Its remaining arguments conform to those for `scanf()`. Consider the code

```
char   str1[] = "1 2 3 go", str2[100], tmp[100];
int    a, b, c;

sscanf(str1, "%d%d%d%s", &a, &b, &c, tmp);
sprintf(str2, "%s %s %d %d %d\n", tmp, tmp, a, b, c);
printf("%s", str2);
```

The function `sscanf()` takes its input from `str1`. It reads three decimal integers and a string, putting them into `a`, `b`, `c`, and `tmp`, respectively. The function `sprintf()` writes to `str2`. More precisely, it writes characters in memory, beginning at the address `str2`. Its output is two strings and three decimal integers. To see what is in `str2`, we invoke `printf()`. It prints the following on the screen:

```
go go 1 2 3
```

Caution: It is the programmer's responsibility to provide adequate space in memory for the output of `sprintf()`.

Reading from a string is unlike reading from a file in the following sense: If we use `sscanf()` to read from `str1` again, then the input starts at the beginning of the string, not where we left off before.

11.4 The Functions fopen() and fclose()

Abstractly, a file can be thought of as a stream of characters. After a file has been opened, the stream can be accessed with file handling functions in the standard library. In this section, we want to explain the use of `fopen()` and `fclose()`.

Files have several important properties: They have a name. They must be opened and closed. They can be written to, or read from, or appended to. Conceptually, until a file is opened nothing can be done to it. It is like a closed book. When it is opened, we can have access to it at its beginning or end. To prevent accidental misuse, we must tell the system which of the three activities—reading, writing, or appending—we will be performing on it. When we are finished using the file, we close it. Consider the following code:

```
#include <stdio.h>

int main(void)
{
   int    a, sum = 0;
   FILE   *ifp, *ofp;

   ifp = fopen("my_file", "r");      /* open for reading */
   ofp = fopen("outfile", "w");      /* open for writing */
   .....
```

This opens two files in the current directory: *my_file* for reading and *outfile* for writing. (The identifier `ifp` is mnemonic for "infile pointer," and the identifier `ofp` is mnemonic for "outfile pointer.") After a file has been opened, the file pointer is used exclusively in all references to the file. Suppose that *my_file* contains integers. If we want to sum them and put the result in *outfile*, we can write

```
while (fscanf(ifp, "%d", &a) == 1)
   sum += a;
fprintf(ofp, "The sum is %d.\n", sum);
```

Note that `fscanf()`, like `scanf()`, returns the number of successful conversions. After we have finished using a file, we can write

```
fclose(ifp);
```

This closes the file pointed to by `ifp`.

A function call of the form `fopen(`*filename, mode*`)` opens the named file in a particular mode and returns a file pointer. There are a number of possibilities for the mode.

Mode	Meaning
`"r"`	open text file for reading
`"w"`	open text file for writing
`"a"`	open text file for appending
`"rb"`	open binary file for reading
`"wb"`	open binary file for writing
`"ab"`	open binary file for appending

Each of these modes can end with a + character. This means that the file is to be opened for both reading and writing.

Mode	Meaning
`"r+"`	open text file for reading and writing
`"w+"`	open text file for writing and reading
`.....`	

Opening for reading a file that cannot be read, or does not exist, will fail. In this case `fopen()` returns a `NULL` pointer. Opening a file for writing causes the file to be created if it does not exist and causes it to be overwritten if it does. Opening a file in append mode causes the file to be created if it does not exist and causes writing to occur at the end of the file if it does.

A file is opened for updating (both reading and writing) by using a + in the mode. However, between a read and a write or a write and a read there must be an intervening call to `fflush()` to flush the buffer, or a call to one of the file positioning function calls `fseek()`, `fsetpos()`, or `rewind()`.

In some operating systems, including UNIX, there is no distinction between binary and text files, except in their contents. The file mechanism is the same for both types of files. In MS-DOS and other operating systems, there are different file mechanisms for each of the two types of files. (See exercise 22, on page 549, for further discussion.)

A detailed description of file handling functions such as `fopen()` and `fclose()` can be found in Section A.12, "Input/Output: `<stdio.h>`," on page 655. Consult the appendix as necessary to understand how the various functions are used.

11.5 An Example: Double Spacing a File

Let us illustrate the use of some file handling functions by writing a program to double-space a file. In `main()`, we open files for reading and writing that are passed as command line arguments. After the files have been opened, we invoke `double_space()` to accomplish the task of double spacing.

In file dbl_space.c

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

void   double_space(FILE *, FILE *);
void   prn_info(char *);

int main(int argc, char **argv)
{
   FILE   *ifp, *ofp;

   if (argc != 3) {
      prn_info(argv[0]);
      exit(1);
   }
   ifp = fopen(argv[1], "r");      /* open for reading */
   ofp = fopen(argv[2], "w");      /* open for writing */
   double_space(ifp, ofp);
   fclose(ifp);
   fclose(ofp);
   return 0;
}

void double_space(FILE *ifp, FILE *ofp)
{
   int  c;

   while ((c = getc(ifp)) != EOF) {
      putc(c, ofp);
      if (c == '\n')
       putc('\n', ofp);   /* found a newline - duplicate it */
   }
}
```

```
void prn_info(char *pgm_name)
{
   printf("\n%s%s%s\n\n%s%s\n\n",
      "Usage:  ", pgm_name, "  infile  outfile",
      "The contents of infile will be double-spaced ",
      "and written to outfile.");
}
```

Suppose we have compiled this program and put the executable code in the file *dbl_space*. When we give the command

dbl_space file1 file2

the program will read from *file1* and write to *file2*. The contents of *file2* will be the same as *file1*, except that every newline character will have been duplicated.

Dissection of the *dbl_space* Program

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

void   double_space(FILE *, FILE *);
void   prn_info(char *);
```

We have included *stdlib.h* because it contains the function prototype for `exit()`, which gets used in `prn_info()`. The identifier `FILE` is a structure defined in *stdio.h*. To make use of files, we do not need to know system-implementation details of how the file mechanism works. The function prototype for `double_space()` shows that it takes two file pointers as arguments.

```
int main(int argc, char **argv)
{
   FILE   *ifp, *ofp;

   if (argc != 3) {
      prn_info(argv[0]);
      exit(1);
   }
```

The identifiers `ifp` and `ofp` are file pointers. More explicitly, they are of type pointer to `FILE`. The program is designed to access two files entered as command line arguments. If there are too few or too many command line arguments, `prn_info()` is invoked to

print information about the program and `exit()` is invoked to exit the program. By convention, `exit()` returns a nonzero value when something has gone wrong.

```
   ifp = fopen(argv[1], "r");      /* open for reading */
   ofp = fopen(argv[2], "w");      /* open for writing */
```

We can think of `argv` as an array of strings. The function `fopen()` is used to open the file named in `argv[1]` for reading. The pointer value returned by the function is assigned to `ifp`. In a similar fashion, the file named in `argv[2]` is opened for writing.

```
   double_space(ifp, ofp);
```

The two file pointers are passed as arguments to `double_space()`, which then does the work of double spacing. One can see that other functions of this form could be written to perform whatever useful work on files was needed.

```
   fclose(ifp);
   fclose(ofp);
```

The function `fclose()` from the standard library is used to close the files pointed to by `ifp` and `ofp`. It is good programming style to close files explicitly in the same function in which they were opened. Any files not explicitly closed by the programmer are closed automatically by the system on program exit.

```
void double_space(FILE *ifp, FILE *ofp)
{
   int  c;
```

The identifier `c` is an `int`. Although it will be used to store characters obtained from a file, eventually it will be assigned the value `EOF`, which is not a character.

```
while ((c = getc(ifp)) != EOF) {
   putc(c, ofp);
   if (c == '\n')
      putc('\n', ofp);   /* found a newline - duplicate it */
}
```

The macro `getc()` reads a character from the file pointed to by `ifp` and assigns the value to `c`. If the value of `c` is not `EOF`, then `putc()` is used to write `c` into the file pointed to by `ofp`. If `c` is a newline character, another newline character is written into the file as well, double spacing the output file. This process continues repeatedly until an `EOF` is encountered. The macros `getc()` and `putc()` are defined in *stdio.h*.

11.6 Using Temporary Files and Graceful Functions

In ANSI C, the programmer can invoke the library function `tmpfile()` to create a temporary binary file that will be removed when it is closed or on program exit. The file is opened for updating with the mode "wb+". In MS-DOS, a binary file can also be used as a text file. In UNIX, binary and text files are the same. In this section, we write an elementary program that illustrates the use of `tmpfile()` and a graceful version of `fopen()`.

The name of our program is *dbl_with_caps*. First, it reads the contents of a file into a temporary file, capitalizing any letters as it does so. Then the program adds the contents of the temporary file to the bottom of the first file.

In file dbl_with_caps.c

```
/* Replicate a file with caps. */

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

FILE   *gfopen(char *filename, char *mode);

int main(int argc, char **argv)
{
   int    c;
   FILE   *fp, *tmp_fp;

   if (argc != 2) {
      fprintf(stderr, "\n%s%s%s\n\n%s\n\n",
       "Usage:  ", argv[0], "  filename",
       "The file will be doubled and some letters capitalized.");
      exit(1);
   }
   fp = gfopen(argv[1], "r+");
   tmp_fp = tmpfile();
   while ((c = getc(fp)) != EOF)
      putc(toupper(c), tmp_fp);
   rewind(tmp_fp);
   fprintf(fp, "---\n");
   while ((c = getc(tmp_fp)) != EOF)
      putc(c, fp);
   return 0;
}
```



```
FILE *gfopen(char *filename, char *mode)
{
   FILE   *fp;

   if ((fp = fopen(filename, mode)) == NULL) {
      fprintf(stderr, "Cannot open %s - bye!\n", filename);
      exit(1);
   }
   return fp;
}
```

Before we explain the program, let us see its effects. Suppose that in file *apple* we have the line

```
A is for apple and alphabet pie.
```

After we give the command

```
dbl_with_caps apple
```

the contents of the file will be

```
A is for apple and alphabet pie.
---
A IS FOR APPLE AND ALPHABET PIE.
```

Dissection of the *dbl_with_caps* Program

■ `fp = gfopen(argv[1], "r+");`

We are using a graceful version of `fopen()` to open a file for both reading and writing. If for some reason the file cannot be opened, a message will be printed and the program exited.

■ `tmp_fp = tmpfile();`

ANSI C provides the function `tmpfile()` to open a temporary file. The file mode is "wb+". On program exit, the file will be removed by the system. (See Section A.12, "Input/Output: <stdio.h>," on page 656, for the function prototype and other details.)

```
while ((c = getc(fp)) != EOF)
   putc(toupper(c), tmp_fp);
```

The macros `getc()` and `putc()` are defined in *stdio.h*. They are being used to read from one file and to write to another. The function prototype for `toupper()` is given in *ctype.h*. If `c` is a lowercase letter, `toupper(c)` returns the corresponding uppercase letter; otherwise, it returns `c`. *Caution:* Some ANSI C compilers do not get this right; we hope that they will improve with time. (See exercise 12, on page 400, in Chapter 8, "The Preprocessor," for further discussion.) You may have to write

```
while ((c = getc(fp)) != EOF)
   if (islower(c))
      putc(toupper(c), tmp_fp);
   else
      putc(c, tmp_fp);
```

```
rewind(tmp_fp);
```

This causes the file position indicator for the stream pointed to by `tmp_fp` to be set to the beginning of the file. This statement is equivalent to

```
fseek(tmp_fp, 0, 0);
```

See Section A.12, "Input/Output: <stdio.h>," on page 657, for the function prototypes and for an explanation of `fseek()`.

```
fprintf(fp, "---\n");
while ((c = getc(tmp_fp)) != EOF)
   putc(c, fp);
```

Now we are reading from the stream pointed to by `tmp_fp` and writing to the stream pointed to by `fp`. Note that a call to `rewind()` occurred before the switch from writing to reading on the stream pointed to by `tmp_fp`.

```
FILE *gfopen(char *filename, char *mode)
{
   .....
```

This is a graceful version of `fopen()`. If something goes wrong, a message is printed on the screen and we exit the program. Note that we wrote to `stderr`. In this program, we could just as well have written to `stdout`. However, in other programs that use this function, there is an advantage to writing to `stderr`. (See exercise 1, on page 542.)

11.7 Accessing a File Randomly

The library functions `fseek()` and `ftell()` are used to access a file randomly. An expression of the form

`ftell(`*file_ptr*`)`

returns the current value of the file position indicator. The value represents the number of bytes from the beginning of the file, counting from zero. Whenever a character is read from the file, the system increments the position indicator by 1. Technically, the file position indicator is a member of the structure pointed to by *file_ptr*. *Caution:* The file pointer itself does not point to individual characters in the stream. This is a conceptual mistake that many beginning programmers make.

The function `fseek()` takes three arguments: a file pointer, an integer offset, and an integer that indicates the place in the file from which the offset should be computed. A statement of the form

`fseek(`*file_ptr* `,` *offset*`,` *place*`);`

sets the file position indicator to a value that represents *offset* bytes from *place*. The value for *place* can be 0, 1, or 2, meaning the beginning of the file, the current position, or the end of the file, respectively. *Caution:* The functions `fseek()` and `ftell()` are guaranteed to work properly only on binary files. In MS-DOS, if we want to use these functions, the file should be opened with a binary mode. In UNIX, any file mode will work.

A common exercise is to write a file backwards. In the following program, the prompt to the user is written to `stderr` so that the program will work with redirection. (See exercise 1, on page 542.) We opened the file with mode `"rb"` so that the program will work in both MS-DOS and UNIX.

Here is a program that writes a file backwards:

In file backward.c

```
/* Write a file backwards. */

#include <stdio.h>
#define   MAXSTRING   100

int main(void)
{
   char   fname[MAXSTRING];
   int    c;
   FILE   *ifp;

   fprintf(stderr, "\nInput a filename:  ");
   scanf("%s", fname);
   ifp = fopen(fname, "rb");        /* binary mode for MS_DOS */
   fseek(ifp, 0, SEEK_END);         /* move to end of the file */
   fseek(ifp, -1, SEEK_CUR);        /* back up one character */
   while (ftell(ifp) > 0) {
      c = getc(ifp);                /* move ahead one character */
      putchar(c);
      fseek(ifp, -2, SEEK_CUR);     /* back up two characters */
   }
   return 0;
}
```

11.8 File Descriptor Input/Output

A file descriptor is a nonnegative integer associated with a file. In this section, we describe a set of library functions that are used with file descriptors. Although these functions are not part of ANSI C, they are available on most C systems in both MS-DOS and UNIX. Because of minor differences, care is needed when porting code from UNIX to MS-DOS or vice versa.

File name	Associated file descriptor
standard input	0
standard output	1
standard error	2

Functions in the standard library that use a pointer to FILE are usually buffered. In contrast, functions that use file descriptors may require programmer-specified buffers. Let us illustrate the use of file descriptors with a program that reads from one file and writes to another, changing the case of each letter.

In file change_case.c

```
/* Change the case of letters in a file. */
#include <ctype.h>
#include <fcntl.h>
#include <unistd.h>          /* use io.h in MS_DOS */

#define   BUFSIZE   1024

int main(int argc, char **argv)
{
   char   mybuf[BUFSIZE], *p;
   int    in_fd, out_fd, n;

   in_fd = open(argv[1], O_RDONLY);
   out_fd = open(argv[2], O_WRONLY | O_EXCL | O_CREAT, 0600);
   while ((n = read(in_fd, mybuf, BUFSIZE)) > 0) {
      for (p = mybuf; p - mybuf < n; ++p)
       if (islower(*p))
          *p = toupper(*p);
       else if (isupper(*p))
          *p = tolower(*p);
      write(out_fd, mybuf, n);
   }
   close(in_fd);
   close(out_fd);
   return 0;
}
```

Dissection of the *change_case* Program

```
#include <ctype.h>
#include <fcntl.h>
#include <unistd.h>          /* use io.h in MS-DOS */
```

The header file *fcntl.h* contains symbolic constants that we are going to use. The header file *unistd.h* contains the function prototypes for `open()` and `read()`. In MS-DOS, we would include *io.h* instead.

```
in_fd = open(argv[1], O_RDONLY);
```

The first argument to `open()` is a file name; the second argument specifies how the file is to be opened. If there are no errors, the function returns a file descriptor; otherwise, the value -1 is returned. The identifier `in_fd` is mnemonic for "in file descriptor." On both MS-DOS and UNIX systems, the symbolic constant `O_RDONLY` is given in *fcntl.h*. It is mnemonic for "open for reading only."

```
out_fd = open(argv[2], O_WRONLY | O_EXCL | O_CREAT, 0600);
```

The symbolic constants in *fcntl.h* that are used to open a file can be combined with the bitwise OR operator. Here, we are specifying that the file is to be opened for writing only, that the file is to be opened exclusively (meaning that it is an error if the file already exists), and that the file is to be created if it does not exist. `O_EXCL` gets used only with `O_CREAT`. If the file is created, then the third argument sets the file permissions; otherwise, the argument has no effect. We will explain about file permissions below.

```
while ((n = read(in_fd, mybuf, BUFSIZE)) > 0) {
    .....
```

A maximum of `BUFSIZE` characters are read from the stream associated with `in_fd` and put into `mybuf`. The number of characters read is returned. The body of the `while` loop is executed as long as `read()` is able to get characters from the stream. In the body of the loop the letters in `mybuf` are changed to the opposite case.

```
write(out_fd, mybuf, n);
```

`n` characters in `mybuf` are written to the stream indicated by `out_fd`.

```
close(in_fd);
close(out_fd);
```

This closes the two files. If the files are not explicitly closed by the programmer, the system will close them on program exit.

Caution: This program is not user-friendly. If you give the command

change_case file1 file2

and *file2* already exists, then the file does not get overwritten, which is the behavior we wanted. A better-designed program would say something to the user in this case.

11.9 File Access Permissions

In UNIX, a file is created with associated access permissions. The permissions determine access to the file for the owner, the group, and for others. The access can be read, write, execute, or any combination of these, including none. When a file is created by invoking `open()`, a three-digit octal integer can be used as the third argument to set the permissions. Each octal digit controls read, write, and execute permissions. The first octal digit controls permissions for the user, the second octal digit controls permissions for the group, and the third octal digit controls permissions for others. ("Others" includes everybody.)

Meaning of each octal digit in the file permissions		
Mnemonic	Bit representation	Octal representation
r--	100	04
-w-	010	02
--x	001	01
rw-	110	06
r-x	101	05
-wx	011	03
rwx	111	07

Now, if we pack three octal digits together into one number, we get the file access permissions. The mnemonic representation is the easiest to remember. The first, second, and third group of three letters refers to the user, the group, and others, respectively.

Examples of file access permissions	
Mnemonic	Octal representation
rw-------	0600
rw----r--	0604
rwxr-xr-x	0755
rwxrwxrwx	0777

The permissions `rwxr-xr-x` mean that the owner can read, write, and execute the file; that the group can read and execute the file; and that others can read and execute the file. In UNIX, the mnemonic file access permissions are displayed with the *ls -l* command. In MS-DOS, file permissions exist, but only for everybody.

11.10 Executing Commands from Within a C Program

The library function `system()` provides access to operating system commands. In both MS-DOS and UNIX, the command *date* causes the current date to be printed on the screen. If we want this information printed on the screen from within a program, we can write

```
system("date");
```

The string passed to `system()` is treated as an operating system command. When the statement is executed, control is passed to the operating system, the command is executed, and then control is passed back to the program.

In UNIX, *vi* is a commonly used text editor. Suppose that from inside a program we want to use *vi* to edit a file that has been given as a command line argument. We can write

```
char   command[MAXSTRING];

sprintf(command, "vi %s", argv[1]);
printf("vi on the file %s is coming up ...\n", argv[1]);
system(command);
```

A similar example works in MS-DOS, provided we replace *vi* by an editor that is available on that system.

As a final example, let us suppose we are tired of looking at all those capital letters produced by the *dir* command on our MS-DOS system. We can write a program that interfaces with this command and writes only lowercase letters on the screen.

In file lower_case.c

```
/* Write only lowercase on the screen. */

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

#define   MAXSTRING   100

int main(void)
{
   char   command[MAXSTRING], *tmp_filename;
   int    c;
   FILE   *ifp;

   tmp_filename = tmpnam(NULL);
   sprintf(command, "dir > %s", tmp_filename);
   system(command);
   ifp = fopen(tmp_filename, "r");
   while ((c = getc(ifp)) != EOF)
      putchar(tolower(c));
   remove(tmp_filename);
   return 0;
}
```

First, we use the library function `tmpnam()` to create a temporary file name. Then we invoke `system()` to redirect the output of the *dir* command into the temporary file. Then we print on the screen the contents of the file, changing each uppercase letter to lowercase. Finally, when we are finished using the temporary file, we invoke the library function `remove()` to remove it. (See Section A.12, "Input/Output: <stdio.h>," on page 662, for details about these functions.)

11.11 Using Pipes from Within a C Program

UNIX systems provide `popen()` and `pclose()` to communicate with the operating system. These functions are not available in MS-DOS systems. Suppose we are tired of looking at all those lowercase letters produced by the *ls* command on our UNIX system.

In file upper_case.c

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

int main(void)
{
   int    c;
   FILE   *ifp;

   ifp = popen("ls", "r");
   while ((c = getc(ifp)) != EOF)
      putchar(toupper(c));
   pclose(ifp);
   return 0;
}
```

The first argument to `popen()` is a string that is interpreted as a command to the operating system; the second argument is a file mode, either `"r"` or `"w"`. When the function is invoked, it creates a pipe (hence the name popen) between the calling environment and the system command that is executed. Here, we get access to whatever is produced by the *ls* command. Because access to the stream pointed to by `ifp` is via a pipe, we cannot use file positioning functions. For example, `rewind(ifp)` will not work. We can only access the characters sequentially. A stream opened by `popen()` should be closed by `pclose()`. If the stream is not closed explicitly, then it will be closed by the system on program exit.

11.12 Environment Variables

Environment variables are available in both UNIX and MS-DOS. The following program can be used to print them on the screen:

```
#include <stdio.h>

int main(int argc, char *argv[], char *env[])
{
   int   i;

   for (i = 0; env[i] != NULL; ++i)
      printf("%s\n", env[i]);
   return 0;
}
```

The third argument to `main()` is a pointer to pointer to `char`, which we can think of as an array of pointers to `char`, or as an array of strings. The system provides the strings, including the space for them. The last element in the array `env` is a `NULL` pointer. On our UNIX system, this program prints

```
HOME=/c/c/blufox/center_manifold
SHELL=/bin/csh
TERM=vt102
USER=blufox
.....
```

To the left of the equal sign is the environment variable; to the right of the equal sign is its value, which should be thought of as a string. On our MS-DOS system, this program prints

```
COMSPEC=C:\COMMAND.COM
BASE=d:\base
INCLUDE=d:\msc\include
.....
```

The UNIX system provides a command to display the environment variables, but the command depends on which shell you are running. In the C shell, the command is *printenv*, whereas in the Bourne shell and in MS-DOS, the command is *set*. The output of the command is the same as the output of our program.

By convention, environment variables are usually capitalized. In a C program, we can use the library function `getenv()` to access the value of an environment variable passed as an argument. Here is an example of how `getenv()` can be used:

```
printf("%s%s\n%s%s\n%s%s\n%s%s\n",
   "          Name: ", getenv("NAME"),
   "          User: ", getenv("USER"),
   "         Shell: ", getenv("SHELL"),
   "Home directory: ", getenv("HOME"));
```

The function prototype is provided in *stdlib.h*. If the string passed as an argument is not an environment variable, the `NULL` pointer is returned.

11.13 The C Compiler

There are many C compilers, and an operating system may provide any number of them. Here are just a few of the possibilities:

Command	The C compiler that gets invoked
cc	The system supplied native C compiler
acc	An early version of an ANSI C compiler from Sun Microsystems
bc	Borland C/C++ compiler, integrated environment
bcc	Borland C/C++ compiler, command line version
gcc	GNU C compiler from the Free Software Foundation
hc	High C compiler from Metaware
occ	Oregon C compiler from Oregon Software
qc	Quick C compiler from Microsoft
tc	Turbo C compiler, integrated environment, from Borland
tcc	Turbo C compiler, command line version, from Borland

In this section, we discuss some of the options that can be used with the *cc* command on UNIX systems. Other compilers provide similar options.

If a complete program is contained in a single file, say *pgm.c*, then the command

```
cc pgm.c
```


translates the C code in *pgm.c* into executable object code and writes it in the file *a.out*. (In MS-DOS, the executable file is *pgm.exe*.) The command *a.out* executes the program. The next *cc* command will overwrite whatever is in *a.out*. If we give the command

```
cc -o pgm pgm.c
```

then the executable code will, instead, be written directly into the file *pgm*; whatever is in *a.out* will not be disturbed.

The *cc* command actually does its work in three stages: First, the preprocessor is invoked, then the compiler, and finally the loader. The loader, or linker, is what puts all the pieces together to make the final executable file. The *-c* option can be used to compile only—that is, to invoke the preprocessor and the compiler, but not the loader. This is useful if we have a program written in more than one file. Consider the command

```
cc -c main.c file1.c file2.c
```

If there are no errors, corresponding object files ending in *.o* will be created. To create an executable file, we can compile a mixture of *.c* and *.o* files. Suppose, for example, that we have an error in *main.c*. We can correct the error in *main.c* and then give the command

```
cc -o pgm main.c file1.o file2.o
```

The use of *.o* files in place of *.c* files reduces compilation time.

Some useful options to the compiler	
`-c`	Compile only, generate corresponding *.o* files.
`-g`	Generate code suitable for the debugger.
`-o` *name*	Put executable output code in *name*.
`-p`	Generate code suitable for the profiler.
`-v`	Verbose option, generates a lot of information.
`-D` *name=def*	Place at the top of each *.c* file the line.
	`#define` *name def*.
`-E`	Invoke the preprocessor but not the compiler.
`-I` *dir*	Look for `#include` files in the directory *dir*.
`-M`	Make a makefile.
`-O`	Attempt code optimization.
`-S`	Generate assembler code in corresponding *.s* files.

Your compiler may not support all of these options, and it may provide others. Or it may use different flag characters. Compilers in MS-DOS usually support different memory models. Consult the documentation for your compiler for a detailed list of options. *Suggestion:* If you have never tried the *-v* option, do so. Some compilers produce a ton of information, and this information can be very useful when you are trying to understand all the details about the compilation process.

11.14 Using the Profiler

In UNIX, the *-p* option used with *cc* causes the compiler to produce code that counts the number of times each routine is called. If the compiler creates an executable file, then it is arranged so that the library function `monitor()` is automatically invoked and a file *mon.out* is created. The file *mon.out* is then used by the *prof* command to generate an execution profile.

As an example of how this all works, suppose we want an execution profile of the *quicksort* routine that we wrote in Section 8.5, "An Example: Sorting with `qsort()`," on page 372. Here is our function `main()`:

In file main.c

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

#define   N   50000

void   quicksort(int *, int *);
```



```
int main(void)
{
   int    a[N], i;

   srand(time(NULL));
   for (i = 0; i < N; ++i)
      a[i] = rand() % 10000;
   quicksort(a, a + N - 1);
   for (i = 0; i < N - 1; ++i)
      if (a[i] > a[i + 1]) {
         printf("SORTING ERROR - bye!\n");
         exit(1);
      }
   return 0;
}
```

To obtain an execution profile of our program, we first compile it with the *-p* option:

```
cc -p -o quicksort main.c quicksort.c
```

Next, we give the command

```
quicksort
```

This causes the file *mon.out* to be created. Finally, to get an execution profile, we give the command

```
prof quicksort
```

This causes the following to be printed on the screen:

```
%time  cumsecs  #call  ms/call  name
 46.9     7.18   9931     0.72  _partition
 16.1     9.64      1  2460.83  _main
 11.7    11.43  19863     0.09  _find_pivot
 10.8    13.08                 mcount
  6.9    14.13  50000     0.02  _rand
  6.4    15.12  19863     0.05  _quicksort
  1.4    15.33                 _monstartup
  0.0    15.33      1     0.00  _gettimeofday
  0.0    15.33      1     0.00  _profil
  0.0    15.33      1     0.00  _srand
  0.0    15.33      1     0.00  _time
```

Not all the named functions are user-defined; some of them, such as `_gettimeofday`, are system routines. An execution profile such as this can be very useful when working to improve execution time efficiency.

11.15 Libraries

Many operating systems provide a utility to create and manage libraries. In UNIX, the utility is called the archiver, and it is invoked with the *ar* command. In the MS-DOS world, this utility is called the librarian, and it is an add-on feature. The Microsoft librarian, for example, is *lib*, whereas the Turbo C librarian is *tlib*. By convention, library files end in *.a* in UNIX, and end in *.lib* in MS-DOS. We will discuss the situation as it pertains to UNIX, but the general ideas apply to any librarian.

In UNIX, the standard C library is usually in the file */lib/libc.a*, but the standard library can exist wholly or in part in other files as well. If UNIX is available to you, try the command

```
ar t /usr/lib/libc.a
```

The key *t* is used to display titles, or names, of files in the library. There are more titles than you care to look at. To count them, you can give the command

```
ar t /usr/lib/libc.a | wc
```

This pipes the output of the *ar* command to the input of the *wc* command, causing lines, words, and characters to be counted. (The name *wc* stands for "word count.") It is not too surprising that the standard library grows with time. On a DEC VAX 11/780 from the 1980s, the standard library contained 311 object files. On a Sun machine that was relatively new in 1990, the standard library contained 498 object files. On the Sun machine that we happen to be using today, the number of object files is 563.

Let us illustrate how programmers can create and use libraries of their own. We will do this in the context of creating a "graceful library." In Section 11.6, "Using Temporary Files and Graceful Functions," on page 511, we presented `gfopen()`, a graceful version of `fopen()`. In a directory named *g_lib*, we have 11 such graceful functions, and we continue to add more from time to time. These are functions such as `gfclose()`, `gcalloc()`, `gmalloc()`, and so forth. Each is written in a separate file, but for the purpose of building a library, they could just as well be in one file. To reinforce the idea of these functions, we give the code for `gcalloc()`:


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

void *gcalloc(int n, unsigned sizeof_something)
{
   void   *p;

   if ((p = calloc(n, sizeof_something)) == NULL) {
      fprintf(stderr, "\nERROR:  calloc() failed - bye.\n\n");
      exit(1);
   }
   return p;
}
```

To create our library, we must first compile the *.c* files to obtain corresponding *.o* files. After we have done this, we give the two commands

```
ar ruv g_lib.a gfopen.o gfclose.o gcalloc.o ...
ranlib g_lib.a
```

The keys *ruv* in the first command stand for replace, update, and verbose, respectively. This command causes the library *g_lib.a* to be created if it does not already exist. If it does exist, then the named *.o* files replace those of the same name already in the library. If any of the named *.o* files are not in the library, they are added to it. The *ranlib* command is used to randomize the library in a form that is useful for the loader.

Now suppose we are writing a program that consists of *main.c* and two other *.c* files. If our program invokes `gfopen()`, we need to make our library available to the compiler. The following command does this:

```
cc -o pgm main.c file1.c file2.c g_lib.a
```

If our program invokes a function and does not supply its function definition, then it will be searched for, first in *g_lib.a* and then in the standard library. Only those functions that are needed will be loaded into the final executable file.

If we write lots of programs, each consisting of many files, then each of the programs should be written in its own directory. Also, we should have a separate directory for our libraries such as *g_lib.a*, and another directory for associated header files such as *g_lib.h*. For each function in *g_lib.a*, we put its prototype in *g_lib.h*. This header file then gets included where needed. To manage all of this, we use the *make* utility. (See Section 11.17, "The Use of make," on page 532.)

11.16 How to Time C Code

Most operating systems provide access to the underlying machine's internal clock. In this section, we show how to use some timing functions. Because our functions are meant to be used in many programs, we put them into the library *u_lib.a*, our utility library. In Section 11.17, "The Use of make," on page 532, we will discuss a program that uses functions from both of our libraries *g_lib.a* and *u_lib.a*.

Access to the machine's clock is made available in ANSI C through a number of functions whose prototypes are in *time.h*. This header file also contains a number of other constructs, including the type definitions for `clock_t` and `time_t`, which are useful for dealing with time. Typically, the two type definitions are given by

```
typedef   long   clock_t;
typedef   long   time_t;
```

and in turn, these types are used in the function prototypes. Here are the prototypes for the three functions we will use in our timing routines:

```
clock_t   clock(void);
time_t    time(time_t *p);
double    difftime(time_t time1, time_t time0);
```

When a program is executed, the operating system keeps track of the processor time that is being used. When `clock()` is invoked, the value returned is the system's best approximation to the time used by the program up to that point. The clock units can vary from one machine to another. The macro

```
#define   CLOCKS_PER_SEC   60        /* machine-dependent */
```

is provided in *time.h*. It can be used to convert the value returned by `clock()` to seconds. *Caution:* In preliminary versions of ANSI C, this macro was called `CLK_TCK`.

The function `time()` returns the number of seconds that have elapsed since 1 January 1970. Other units and other starting dates are possible, but these are the ones typically used. If the pointer argument passed to `time()` is not `NULL`, then the value returned gets assigned to the variable pointed to as well. One typical use is

```
srand(time(NULL));
```

This seeds the random-number generator. If two values produced by `time()` are passed to `difftime()`, the difference expressed in seconds is returned as a `double`.

We want to present a set of timing routines that can be used for many purposes, including the development of efficient code. We keep these functions in the file *time_keeper.c*.

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

#define   MAXSTRING   100

typedef struct {
   clock_t   begin_clock, save_clock;
   time_t    begin_time, save_time;
} time_keeper;

static time_keeper   tk;                  /* known only to this file */

void start_time(void)
{
   tk.begin_clock = tk.save_clock = clock();
   tk.begin_time = tk.save_time = time(NULL);
}

double prn_time(void)
{
   char      s1[MAXSTRING], s2[MAXSTRING];
   int       field_width, n1, n2;
   double    clocks_per_second = (double) CLOCKS_PER_SEC,
             user_time, real_time;

   user_time = (clock() - tk.save_clock) / clocks_per_second;
   real_time = difftime(time(NULL), tk.save_time);
   tk.save_clock = clock();
   tk.save_time = time(NULL);

   /* print the values found, and do it neatly */

   n1 = sprintf(s1, "%.1f", user_time);
   n2 = sprintf(s2, "%.1f", real_time);
   field_width = (n1 > n2) ? n1 : n2;
   printf("%s%*.1f%s\n%s%*.1f%s\n\n",
      "User time: ", field_width, user_time, " seconds",
      "Real time: ", field_width, real_time, " seconds");
   return user_time;
}
```

Note that the structure `tk` is external to the functions and is known only in this file. It is used for communication between the functions. When `start_time()` is invoked, the values returned from `clock()` and `time()` are stored in `tk`. When `prn_time()` is invoked, new values from `clock()` and `time()` are used to compute and print the elapsed user time and the elapsed real time, and new values are stored in `tk`. User time is whatever the system allocates to the running of the program; real time is wall-clock time. In a time-shared system they need not be the same.

The function `prn_total_time()` is also in the file, but is not shown. It is similar to `prn_time()`, except that the elapsed times are computed relative to the last invocation of `start_time()` rather than the last invocation of any of the three functions.

Because our timing routines are meant to be used in a variety of programs, we put them into *u_lib.a*, our utility library. The following commands do this:

```
cc -c time_keeper.c;
ar ruv u_lib.a time_keeper.o;
ranlib u_lib.a
```

In the header file *u_lib.h*, we put the prototypes of the functions in *u_lib.a*. The header file then gets included elsewhere as needed.

Now we demonstrate how our timing routines can be used. In certain applications, fast floating-point multiplication is desired. Should we use variables of type `float` or `double`? The following program can be used to test this:

In file mult_time.c

```
/* Compare float and double multiplication times. */

#include <stdio.h>
#include "u_lib.h"

#define   N   100000000     /* one hundred million */
```



```
int main(void)
{
   long     i;
   float    a, b = 3.333, c = 5.555;   /* arbitrary values */
   double   x, y = 3.333, z = 5.555;

   printf("Number of multiplies: %d\n\n", N);
   printf("Type float:\n\n");
   start_time();
   for (i = 0; i < N; ++i)
      a = b * c;
   prn_time();
   printf("Type double:\n\n");
   for (i = 0; i < N; ++i)
      x = y * z;
   prn_time();
   return 0;
}
```

On an older machine with a traditional C compiler, we find, much to our surprise, that single-precision multiplication is *slower* than double-precision multiplication! In traditional C, any `float` is automatically promoted to a `double`. Perhaps the results are due to the time it takes the machine to do the conversion. On another machine with an ANSI C compiler, we find that single-precision multiplication is about 30 percent faster. This is in line with what we expected. When we try the program on a Sun Sparcstation 10, we obtained the following results:

```
Number of multiplies: 100000000
Type float:
User time: 33.6 seconds
Real time: 34.0 seconds
Type double:
User time: 33.5 seconds
Real time: 33.0 seconds
```

Once again we are surprised! We expected that multiplication with `floats` would yield a 30 percent saving in time, but that is not the case. Note that the real time listed for `doubles` is less than the user time. This happens because the clocking mechanism as a whole is only approximate. *Caution:* To get an accurate measure of machine multiplication time, the overhead of the `for` loops themselves, as well as general program overhead (relatively negligible), has to be taken into account.

11.17 The Use of *make*

For both the programmer and the machine, it is inefficient and costly to keep entirely in one file a moderate or large size program that has to be recompiled repeatedly. A much better strategy is to write the program in multiple *.c* files, compiling them separately as needed. The *make* utility can be used to keep track of source files and to provide convenient access to libraries and their associated header files. This powerful utility is always available in UNIX and often available in MS-DOS, where it is an add-on feature. Its use greatly facilitates both the construction and the maintenance of programs.

Let us suppose we are writing a program that consists of a number of *.h* and *.c* files. Typically, we would place all these files in a separate directory. The *make* command reads a file whose default name is *makefile*. This file contains the dependencies of the various modules, or files, making up the program, along with appropriate actions to be taken. In particular, it contains the instructions for compiling, or recompiling, the program. Such a file is called a *makefile*.

For simplicity, let us imagine that we have a program contained in two files, say *main.c* and *sum.c*, and that a header file, say *sum.h*, is included in each of the *.c* files. We want the executable code for this program to be in the file *sum*. Here is a simple makefile that can be used for program development and maintenance:

```
sum: main.o sum.o
      cc -o sum main.o sum.o
main.o: main.c sum.h
      cc -c main.c
sum.o: sum.c sum.h
      cc -c sum.c
```

The first line indicates that the file *sum* depends on the two object files *main.o* and *sum.o*. It is an example of a *dependency* line; it must start in column 1. The second line indicates how the program is to be compiled if one or more of the *.o* files have been changed. It is called an *action* line or a *command*. There can be more than one action following a dependency line. A dependency line and the action lines that follow it make up what is called a *rule*. *Caution:* Each action line must begin with a tab character. On the screen, a tab character looks like a sequence of blanks.

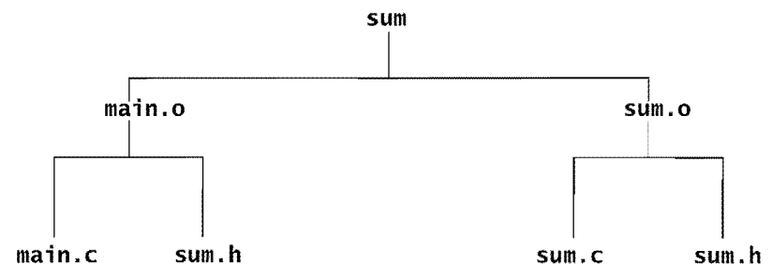
By default, the *make* command will make the first rule that it finds in the makefile. But dependent files in that rule may themselves be dependent on other files as specified in other rules, causing the other rules to be made first. These files, in turn, may cause yet other rules to be made.

The second rule in our makefile states that *main.o* depends on the two files *main.c* and *sum.h*. If either of these two files is changed, then the action line shows what must be done to update *main.o*. After this makefile has been created, the programmer can compile or recompile the program *sum* by giving the command

```
make
```

With this command, *make* reads the file *makefile*, creates for itself a dependency tree, and takes whatever action is necessary.

The dependency tree used internally by make



Certain rules are built into *make*, including the rule that a *.o* file depends on the corresponding *.c* file. Because of this, an equivalent makefile is given by

```
sum: main.o sum.o
      cc -o sum main.o sum.o
main.o: sum.h
      cc -c main.c
sum.o: sum.h
      cc -c sum.c
```

The *make* utility recognizes a number of built-in macros. Using one of them, we get yet another equivalent makefile:

```
sum: main.o sum.o
      cc -o sum main.o sum.o
main.o sum.o: sum.h
      cc -c $*.c
```

Here, the second rule states that the two *.o* files depend on *sum.h*. If we edit *sum.h*, then both *main.o* and *sum.o* must be remade. The macro `$*.c` expands to `main.c` when *main.o* is being made, and it expands to `sum.c` when *sum.o* is being made.

A makefile consists of a series of entries called *rules* that specify dependencies and actions. A rule begins in column 1 with a series of blank-separated target files, followed by a colon, followed by a blank-separated series of prerequisite files, also called *source* files. All the lines beginning with a tab that follow this are the actions—such as compilation—to be taken by the system to update the target files. The target files are dependent in some way on the prerequisite files and must be updated when the prerequisite files are modified.

In exercise 33, on page 406 in Chapter 8, "The Preprocessor," we suggested that a serious comparison of `qsort()` and `quicksort()` be undertaken. We wrote a program to do this that compared the running times for three sorting routines: our own `quicksort()`, the system-supplied `qsort()`, and a `qsort()` from another system for which we were able to borrow the code. The borrowed `qsort()` consisted of some 200 lines of code in a single file. Including this file, our program consisted of approximately 400 lines written in five files. Here is the makefile that we used:

In file makefile

```
# Makefile to compare sorting routines.
BASE    =  /home/blufox/base
CC      =  gcc
CFLAGS  =  -O -Wall
EFILE   =  $(BASE)/bin/compare_sorts
INCLS   =  -I$(LOC)/include
LIBS    =  $(LOC)/lib/g_lib.a  \
           $(LOC)/lib/u_lib.a
LOC     =  /usr/local
OBJS  =  main.o      another_qsort.o   chk_order.o  \
         compare.o   quicksort.o
$(EFILE): $(OBJS)
      @echo "linking ..."
      @$(CC)  $(CFLAGS)  -o $(EFILE)  $(OBJS)  $(LIBS)
$(OBJS): compare_sorts.h
      $(CC)  $(CFLAGS)  $(INCLS)  -c  $*.c
```

Dissection of the Makefile for the *compare_sorts* Program

■ `# Makefile to compare sorting routines.`

Comments can be put in a makefile. A comment begins with a # and extends to the end of the line.

■ `BASE    =  /home/blufox/base`

This is an example of a macro definition. The general form of a macro definition is

macro_name = *replacement_string*

By convention, macro names are usually capitalized, but they do not have to be. The replacement string can contain white space. If a backslash \ occurs at the end of the line, then the replacement string continues to the next line. Our home directory on this particular machine is `/home/blufox`. We created the subdirectory `base` to hold all our other major subdirectories. We think of it as our "base of operation." The macro `BASE` refers to this subdirectory.

■
```
CC      =  gcc
CFLAGS  =  -O -Wall
```

The first macro specifies the C compiler that we are using, in this case the GNU C compiler. The second macro specifies the options, or "C flags," that will be used when *gcc* gets invoked. The *-O* option turns on the optimizer; the *-Wall* option asks for all warnings. If we had written

```
CC      =  gcc
CFLAGS  =
```

instead, then the replacement string for `CFLAGS` would be empty.

■ `EFILE   =  $(BASE)/bin/compare_sorts`

The macro `EFILE` specifies where we want to put our executable file. Macro evaluation, or invocation, occurs with a construct of the form `$(`*macro_name*`)`. This produces the string value (possibly empty) of *macro_name*. The string value of the macro is also called its *replacement string*. Thus

```
EFILE   =  $(BASE)/bin/compare_sorts
```

is equivalent to

```
EFILE   =  /home/blufox/base/bin/compare_sorts
```

```
■  INCLS   =  -I$(LOC)/include
   LIBS    =  $(LOC)/lib/g_lib.a  \
              $(LOC)/lib/u_lib.a
   LOC     =  /usr/local
```

The first macro specifies the `-I` option followed directly by the name of a directory that contains include files. The second macro specifies two libraries. The third macro specifies another directory. Note that a backslash is used to continue to the next line. The first library, *g_lib.a*, is the graceful library, which we discussed in Section 11.15, "Libraries," on page 526. The second library, *u_lib.a*, is the utility library, which we discussed in Section 11.16, "How to Time C Code," on page 530. Because our program invokes functions from these libraries, the libraries must be made available to the compiler. We keep the associated header files *g_lib.h* and *u_lib.h* in the directory `$(LOC)/include`. The compiler will have to be told to look in this directory for header files. Note that a macro can be evaluated before it has been defined.

```
■  OBJS  =  main.o       another_qsort.o   chk_order.o  \
           compare.o    quicksort.o
```

The macro `OBJS` is defined to be the list of object files that occurs on the right side of the equal sign. Note that we used a backslash to continue the line. Although we put *main.o* first and then list the other *.o* files alphabetically, order is unimportant.

```
■  $(EFILE): $(OBJS)
        @echo "linking ..."
        @$(CC)  $(CFLAGS)  -o $(EFILE)  $(OBJS)  $(LIBS)
```

The first line is a dependency line. The second and third lines specify the actions to be taken. Note carefully that action lines begin with a single tab character. (It looks like eight blank spaces on the screen.) The `@` symbol means that the action line itself is not to be echoed on the screen. (See exercise 28, on page 552.) Because macro invocation has the form

$(macro_name)

the construct `$(EFILE)` is replaced by

```
$(BASE)/bin/compare_sorts
```

which in turn is replaced by

```
/home/blufox/base/bin/compare_sorts
```


Similarly, `$(OBJS)` is replaced by the list of object files, and so forth. Thus, the dependency line states that the executable file depends on the object files. If one or more of the object files has been updated, then the specified actions occur. The second action line is expanded to

```
@gcc  -O -Wall  -o /home/blufox/base/bin/compare_sorts  main.o  \
   another_qsort.o  chk_order.o  compare.o  quicksort.o        \
   /home/blufox/base/lib/g_lib.a  /home/blufox/base/lib/u_lib.a
```

Although we have written it on three lines because of space limitations on the printed page, this is actually generated as a single line. *Suggestion:* If *make* is new to you, build your makefiles without the `@` symbol at first. Later, after you understand its effects, you can use the `@` symbol to prevent echoing.

```
■  $(OBJS): compare_sorts.h
        $(CC)  $(CFLAGS)  $(INCLS)  -c  $*.c
```

The first line is a dependency line; it says that all the object files depend on the header file *compare_sorts.h*. If the header file has been updated, then all the object files have to be updated, too. This is done through the action line. In UNIX, action lines *must* begin with a tab. In MS-DOS, they can start with a tab or with one or more blanks. The construct `$*` that occurs in the action line is a predefined macro called the *base filename macro*. It expands to the filename being built, excluding any extension. For example, if *main.o* is being built, then `$*.c` expands to `main.c`, and the action line becomes

```
gcc  -O -Wall  -I/usr/local/include  -c  main.c
```

Certain dependencies are built into the *make* utility. For example, each *.o* file depends on the corresponding *.c* file. This means that if a *.c* file is changed, then it will be recompiled to produce a new *.o* file, and this in turn will cause all the object files to be relinked.

```
■  -I/usr/local/include
```

An option of the form `-I`*dir* means "look in the directory *dir* for `#include` files." This option complements our use of libraries. At the top of the *.c* files making up this program, we have the line

```
#include "compare_sorts.h"
```

and at the top of *compare_sorts.h* we have the lines

```
#include "g_lib.h"
#include "u_lib.h"
```

These header files contain the function prototypes for the functions in our libraries. The `-I` option tells the compiler where to find these header files.

The *make* utility can be used to maintain programs in any language, not just C and C++. More generally, *make* can be used in any kind of project that consists of files with dependencies and associated actions.

11.18 The Use of *touch*

The *touch* utility is always available in UNIX and is often available in MS-DOS. It puts a new time on a file. The *make* utility decides which actions to take by comparing file times, and *touch* can be used to direct what *make* does.

To illustrate the use of *touch*, let us assume we have the makefile discussed in the previous section, along with the relevant *.h*, *.c*, and *.o* files. To put the current date on the file *compare_sorts.h*, we can give the command

```
touch compare_sorts.h
```

This causes the file to have a more recent time than all the object files that depend on it. Now, if we give the command

```
make
```

all the *.c* files will be recompiled and linked to create a new executable file.

11.19 Other Useful Tools

Operating systems provide many useful tools for the programmer. Here, we will list a few of the tools found on UNIX systems, along with some remarks. Comparable utilities are sometimes available in MS-DOS.

Command	Remarks
`cb`	The C beautifier; it can be used to "pretty print" C code.
`dbx`	A source-level debugger; code must be compiled with the `-g` option.
`diff`	Prints the lines that differ in two files.
`gdb`	The GNU debugger; code must be compiled with the `-g` option.
`grep`	Searches for a pattern in one or more files; a major tool for programmers.
`indent`	A C code "pretty printer" with lots of options.
`wc`	Counts lines, words, and characters in one or more files.

The *cb* utility reads from `stdin` and writes to `stdout`. It is not very powerful. To see what it can do, try the command

```
cb < pgm.c
```

where *pgm.c* is poorly formatted. The utility *indent* is more powerful, but unlike *cb*, it is not universally available. To make serious use of *indent*, you will need to read the online manual.

A debugger allows the programmer to step through the code a line at a time and to see the values of variables and expressions at each step. This can be extremely helpful in discovering why a program is not acting as the programmer expected. The programming world is full of debuggers, and *dbx* is not a particularly good one. It just happens to be one that is generally available on UNIX systems. In the MS-DOS world, debuggers are an add-on product. Borland, Microsoft, and others provide excellent products.

Tools such as *diff*, *grep*, and *wc* are of a general nature. They get used by everyone, not just programmers. Although these are UNIX tools, they are often available in MS-DOS as well, especially *grep*, which is very useful to programmers.

Finally, let us mention that C can be used in conjunction with other high-level tools, some of which are languages in their own right.

Utility	Remarks
awk	A pattern scanning and processing language
bison	GNU's version of *yacc*
csh	The C shell, which is programmable
flex	GNU's version of *lex*
lex	Generates C code for lexical analysis
nawk	A newer, more powerful, version of *awk*
perl	Practical extraction and report language
sed	A stream editor that takes its commands from a file
yacc	"Yet another compiler-compiler," used to generate C code

Of particular importance to programmers are *lex* and *yacc*, or the corresponding GNU utilities, *flex* and *bison*; see *The UNIX Programming Environment* by Brian Kernighan and Rob Pike (Englewood Cliffs, N.J.: Prentice-Hall, 1984). The Free Software Foundation produces GNU tools. These tools run on many platforms, and they can be obtained via the Internet. (The Internet provides access to other tools from other places as well.) If you have an Internet connection, the command

ftp prep.ai.mit.edu

will connect you to a machine from which you can download GNU tools. (First-time *ftp* users will need assistance.)

Summary

1 The functions `printf()` and `scanf()`, and the related file and string versions of these functions, all use conversion specifications in a control string to deal with a list of arguments of variable length.

2 The standard header file *stdio.h* is included if files are used. It contains the definitions of the identifier `FILE` (a structure) and the file pointers `stdin`, `stdout`, and `stderr`. It also contains prototypes of many file handling functions and definitions for the macros `getc()` and `putc()`. The function call `getc(ifp)` reads the next character from the file pointed to by `ifp`.

3 To open and close files, we use `fopen()` and `fclose()`, respectively. After a file has been opened, the file pointer is used to refer to the file.

4 A file can be thought of as a stream of characters. The stream can be accessed either sequentially or randomly. When a character is read from a file, the operating system increments the file position indicator by 1.

5 The system opens the three standard files `stdin`, `stdout`, and `stderr` at the beginning of each program. The function `printf()` writes to `stdout`. The function `scanf()` reads from `stdin`. The files `stdout` and `stderr` are usually connected to the screen. The file `stdin` is usually connected to the keyboard. Redirection causes the operating system to make other connections.

6 Files are a scarce resource. The maximum number of files that can be open simultaneously is given by the symbolic constant `FOPEN_MAX` in *stdio.h*. This number is system-dependent; typically, it is in the range from 20 to 100. It is the programmer's responsibility to keep track of which files are open. On program exit, any open files are closed by the system automatically.

7 A set of functions that use file descriptors is available in most systems, even though these functions are not part of ANSI C. They require user-defined buffers. The file descriptors of `stdin`, `stdout`, and `stderr` are 0, 1, and 2, respectively.

8 An operating system command can be executed from within a program by invoking `system()`. In MS-DOS, the statement

```
system("dir");
```

will cause a list of directories and files to be listed on the screen.

9 In UNIX, the function `popen()` can be used to communicate with the operating system. Consider the command

```
wc *.c
```

It prints on the screen the word count of all the *.c* files in the current directory. To get access to this stream of characters from within a program, the programmer can write

```
FILE   *ifp;
ifp = popen("wc *.c", "r");
```

Streams that are opened with `popen()` should be closed with `pclose()`.

10 Many operating systems provide a utility to create and manage libraries. In UNIX, the utility is called the archiver, and it is invoked with the *ar* command. In the MS-DOS world, this utility is called the librarian, and it is an add-on feature.

11 The *make* utility can be used to keep track of source files and to provide convenient access to libraries and associated header files.

Exercises

1 Rewrite the *dbl_space* program in Section 11.5, "An Example: Double Spacing a File," on page 507, so that it gets the name of the input file as a command line argument and writes to `stdout`. After this has been done, the command

```
dbl_sp infile > outfile
```

can be used to double-space whatever is in *infile*, with the output being written into *outfile*. Because the program is intended to be used with redirection, it now makes sense to invoke `fprintf(stderr, ...)` rather than `printf(...)` in `prn_info()`. If the error message is written to `stdout`, it will be redirected; the user will not see the message on the screen. The symbol > is used to redirect whatever is written to `stdout`. It does not affect whatever is written to `stderr`. Try writing the program two ways: with the error message being written first to `stderr` and then to `stdout`.

Experiment with the two versions of the program so that you understand the different effects.

2 Rewrite the *dbl_space* program in Section 11.5, "An Example: Double Spacing a File," on page 507, so that it uses a command line option of the form *-n*, where *n* can be 1, 2, or 3. If *n* is 1, then the output should be single spaced. That is, two or more contiguous newline characters in the input file should be written as a single newline character in the output file. If *n* is 2, then the output file should be strictly double spaced. That is, one or more contiguous newline characters in the input file should be rewritten as a pair of newline characters in the output file. If *n* is 3, the output file should be strictly triple spaced.

3 Write `getstring()` and `putstring()` functions. The first function should use a file pointer, say `ifp`, and the macro `getc()` to read a string from the file pointed to by `ifp`. The second function should use a file pointer, say `ofp`, and the macro `putc()` to write a string to the file pointed to by `ofp`. Write a program to test your functions.

4 Write a program to number the lines in a file. The input file name should be passed to the program as a command line argument. The program should write to `stdout`. Each line in the input file should be written to the output file with the line number and a space prepended.

5 Read about the `ungetc()` function in Section A.12, "Input/Output: <stdio.h>," on page 660. After three characters have been read from a file, can `ungetc()` be used to push three characters back onto the file? Write a program to test this.

6 Write a program that displays a file on the screen 20 lines at a time. The input file should be given as a command line argument. The program should display the next 20 lines after a carriage return has been typed. (This is an elementary version of the *more* utility in UNIX.)

7 Modify the program you wrote in the previous exercise to display one or more files given as command line arguments. Also, allow a command line option of the form *-n* to be used, where *n* is a positive integer specifying the number of lines that are to be displayed at one time. In MS-DOS, the command to clear the screen is *cls*; in UNIX it is *clear*. Try either one or the other of these commands on your system so that you understand its effects. Use either `system("cls")` or `system("clear")` in your program just before you write each set of lines to the screen.

8 The library function `fgets()` can be used to read from a file a line at a time. Read about `fgets()` in Section A.12, "Input/Output: <stdio.h>," on page 659. Write a program called *search* that searches for patterns. If the command

search hello my_file

is given, then the string pattern *hello* is searched for in the file *my_file*. Any line that contains the pattern is printed. (This program is an elementary version of *grep*.) *Hint:* Use the following code:

```
char   line[MAXLINE], *pattern;
FILE   *ifp;
if (argc != 3) {
   .....
}

if ((ifp = fopen(argv[2], "r")) == NULL) {
   fprintf(stderr, "\nCannot open %s\n\n", argv[2]);
   exit(1);
}

pattern = argv[1];
while (fgets(line, MAXLINE, ifp) != NULL) {
   if (strstr(line, pattern) != NULL)
      .....
```

9 Modify the function you wrote in the previous exercise. If the command line option *-n* is present, then the line number should be printed as well.

10 Compile the following program and put the executable code into a file, say *try_me*:

```
#include <stdio.h>

int main(void)
{
   fprintf(stdout, "She sells sea shells\n");
   fprintf(stderr, "by the seashore.\n");
   return 0;
}
```

Execute the program so that you understand its effects. What happens when you redirect the output? Try the command

try_me > tmp

Make sure you read the file *tmp* after you do this. In UNIX, you should also try the command

try_me > & tmp

This causes the output that is written to `stderr` to be redirected, too. Make sure that you look at what is in *tmp*. You may be surprised!

11 Write a program called *wrt_rand* that creates a file of randomly distributed numbers. The filename is to be entered interactively. Your program should use three functions. Here is the first function:

```
void get_info(char *fname, int *n_ptr)
{
   printf("\n%s\n\n%s",
      "This program creates a file of random numbers.",
      "How many random numbers would you like?  ");
   scanf("%d", n_ptr);
   printf("\nIn what file would you like them?  ");
   scanf("%s", fname);
}
```

After this function has been invoked in `main()`, you could write

```
ofp = fopen(fname, "w");
```

However, the named file may already exist; if it does, overwriting it will destroy whatever is in the file currently. In this exercise, we want to write cautious code. If the file already exists, report this fact to the user and ask permission to overwrite the file. Use for the second function in your program the following "careful" version of `fopen()`:

```
FILE *cfopen(char *fname, char *mode)
{
   char   reply[2];
   FILE   *fp;

   if (strcmp(mode, "w") == 0 && access(fname, F_OK) == 0) {
      printf("\nFile exists.  Overwrite it?  ");
      scanf("%1s", reply);
      if (*reply != 'y' && *reply != 'Y') {
         printf("\nBye!\n\n");
         exit(1);
      }
   }
   fp = gfopen(fname, mode);
   return fp;
}
```

(Read about `access()` in Section A.16, "Miscellaneous," on page 680.) The third function is `gfopen()`, the graceful version of `fopen()` that was presented in Section 11.6, "Using Temporary Files and Graceful Functions," on page 511. *Hint:* To write your randomly distributed numbers neatly, use the following code:

```
for (i = 1; i <= n; ++i) {
   fprintf(ofp, "%12d", rand());
   if (i % 6 == 0 || i == n)
      fprintf(ofp, "\n");
}
```

12 Accessing a string is not like accessing a file. When a file is opened, the file position indicator keeps track of where you are in the file. There is no comparable mechanism for a string. Write a program that contains the following lines and explain what gets printed in `tmp1` and `tmp2`:

```
char   c, s[] = "abc", *p = s;
int    i;
FILE   *ofp1, *ofp2;

ofp1 = fopen("tmp1", "w");
ofp2 = fopen("tmp2", "w");
for (i = 0; i < 3; ++i) {
   sscanf(s, "%c", &c);
   fprintf(ofp1, "%c", c);
}
```



```
for (i = 0; i < 3; ++i) {
   sscanf(p++, "%c", &c);
   fprintf(ofp2, "%c", c);
}
```

13 In this exercise, we examine a typical use of `sscanf()`. Suppose we are writing a serious interactive program that asks the user to input a positive integer. To guard against errors, we can pick up as a string the line typed by the user. The following is one way to process the string:

```
char   line[MAXLINE];
int    error, n;

do {
   printf("Input a positive integer:  ");
   fgets(line, MAXLINE, stdin);
   error = sscanf(line, "%d", &n) != 1 || n <= 0;
   if (error)
      printf("\nERROR: Do it again.\n");
} while (error);
```

This will catch some typing errors, but not all. If, for example, `23e` is typed instead of `233`, the error will not be caught. Modify the code so that if anything other than a digit string surrounded by optional white space is typed, the input is considered to be in error. Use these ideas to rewrite the *wrt_rand* program that you wrote in exercise 11, on page 545.

14 The two conversion characters `x` and `X` can be used to print an expression as a hexadecimal number. Are the two conversion characters equivalent? *Hint:* Try

```
printf("11259375 = %#lx\n", 11259375);
printf("11259375 = %#lX\n", 11259375);
```

(In case you are wondering, the number `11259375` was carefully chosen.)

15 Can your compiler handle the conversion character `n` correctly? (A few years ago, all the ones we tried could not, but now they can.) Try the following:

```
int   a, b, c;

printf("a%nb%nc%n %d %d %d\n", &a, &b, &c, a, b, c);
```

16 Can we give flag characters in a conversion specification in any order? The ANSI C document is not too specific about this point, but it seems that the intent is for any order to be acceptable. See what happens with your compiler when you try the following code:

```
printf("%0+17d\n", 1);
printf("%+017d\n", 1);
```

17 Will the following code get a hexadecimal number from a string? What happens if `0x` is deleted in the string?

```
char   s[] = "0xabc";
int    n;

sscanf(s, "%x", &n);
printf("Value of n: %d\n", n);
```

18 Did you read the table of conversion characters for `scanf()` carefully? Does the following code make sense? Explain.

```
char        s[] = "-1";
unsigned    n;

sscanf(s, "%u", &n);
printf("Value of n: %u\n", n);
```

19 Investigate how `tmpnam()` makes its names. Try executing, for example, the following program:

```
#include <stdio.h>

int main(void)
{
   char     tfn[100];     /* tfn = tmp filename */

   tmpnam(tfn);
   printf("1: tfn = %s\n", tfn);
   tmpnam(tfn);
   printf("2: tfn = %s\n", tfn);
   tmpnam(tfn);
   printf("3: tfn = %s\n", tfn);
   return 0;
}
```


Execute the program repeatedly so that you understand its effects. Notice that `tfn` changes one way within the program, and it changes another way with each execution of the program. On our system, the following line occurs in *stdio.h*:

```
#define   TMP_MAX   17576       /* 26 * 26 * 26 */
```

In ANSI C, repeated calls to `tmpnam()` are supposed to generate at least `TMP_MAX` unique names. On our system, exactly `TMP_MAX` unique names are generated. What happens on your system?

20 Is the Borland C/C++ compiler available to you? If so, try the command

bcc

When no files are given with the command, then a list of all the options is printed on the screen. This is a very nice feature. Try it.

21 Our program that double-spaces a file can be invoked with the command

dbl_space infile outfile

If *outfile* exists, then it will be overwritten. This is potentially dangerous. Rewrite the program so it writes to `stdout` instead. Then the program can be invoked with the command

dbl_space infile > outfile

This program design is much safer. Of all the system commands, only a few are designed to overwrite a file. After all, nobody likes to lose a file by accident.

22 In the early days of MS-DOS, a control-z character within the file was used as an end-of-file mark. Although this is not done now, if a file has a control-z in it, and it is opened as a text file for reading, characters beyond the control-z may be inaccessible. Write a program with the following lines in it:

```
char   cntrl_z = '\032';     /* octal escape for control-z */
int    c;
FILE   *ifp, *ofp;
```

```
ofp = fopen("tmp", "w");

fprintf(ofp, "%s%c%s\n",
   "A is for apple", cntrl_z, " and alphabet pie.");
fclose(ofp);
ifp = fopen("tmp", "r");              /* open as a text file */
while ((c = getc(ifp)) != EOF)        /* print the file */
   putchar(c);
fclose(ifp);
printf("\n---\n");                    /* serves as a marker */
ifp = fopen("tmp", "rb");             /* open as a binary file */
while ((c = getc(ifp)) != EOF)        /* print the file */
   putchar(c);
```

What gets printed? (Does the program act differently on a UNIX system?) In MS-DOS, try the command

type tmp

Only the characters before the control-z are printed. How do you know that there are more characters in the file? *Hint:* Try the *dir* command. Normally, control-z characters are not found in text files, but they certainly can occur in binary files. Subtle problems can occur if you open a binary file for processing with mode "r" instead of "rb".

23 If UNIX is available to you, experiment to see what the following program does:

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

#define   MAXSTRING   100

int main(int argc, char **argv)
{
   char   command[MAXSTRING];

   sprintf(command, "sort -r %s", argv[1]);
   system(command);
   return 0;
}
```

Actually, the program needs improvement. Rewrite it so that it prints a prompt to the user. Give the command

man sort

to read about the *sort* utility.

24 If you are a C programmer, should you care about assembler code? Surprisingly, the answer is yes. The *-S* option causes your compiler to produce a *.s* file, and that file can be useful to you, even if you cannot read a line of assembler code. Write a simple program that contains the following lines:

```
int   i = 10;

while (--i != 0)                  /* inefficient? */
   printf("i = %d\n", i);
```

The value of the expression `--i != 0` controls the execution of the `while` loop. Is the control mechanism inefficient? Write another program that contains the lines

```
int   i = 10;

while (--i)                       /* better? */
   printf("i = %d\n", i);
```

Compile both of your programs with the *-S* option. Then look at the difference between the two *.s* files. In UNIX, this can be done with the command

diff pgm1.s pgm2.s

Is your second program more efficient than your first?

25 Is Turbo C available to you? If so, make the following modifications to the *change_case* program we presented in Section 11.8, "File Descriptor Input/Output," on page 515. Include the standard header files *io.h* and *sys\stat.h*. Replace the octal constant `0600` by `S_IREAD | S_IWRITE`. (These symbolic constants are discussed in the Turbo C manual.) With these changes the program should compile and execute. Does it?

26 If UNIX is available to you, give the command

ls -l

This provides a long listing of all the files and subdirectories in the current directory. Note that file modes are displayed. Read about the *chmod* utility in the online manual. To further your understanding about file modes, try commands similar to the following lines which have octal numbers. Are the leading zeros necessary?

```
date > tmp; ls -l tmp
chmod a+rwx tmp; ls -l tmp
chmod 0660 tmp; ls -l tmp
chmod 0707 tmp; ls -l tmp
```

27 Look carefully at the execution profile we presented in Section 11.14, "Using the Profiler," on page 524. You can see that `rand()` was called 50,000 times. This is correct because the size of the array is 50,000. Note that the number of function calls to `find_pivot()` equals the number of calls to `quicksort()`. By looking at the code, we easily convince ourselves that this, too, is correct. But what about the relationship between the number of calls to `partition()` and the number of calls to `quicksort()`? Can you give a precise explanation?

28 The last makefile that we presented in this chapter is a real one. Even though we dissected it, anyone who has not had experience with *make* will find the concepts difficult to grasp. If this utility is new to you, try the *make* command after you have created the following file:

In file makefile

```
# Experiment with the make command.
go: hello date list
      @echo Goodbye!
hello:
      @echo Hello!
date:
      @date; date; date
list:
      @pwd; ls
```

What happens if you remove the `@` characters? What happens if a file named *hello* or *date* exists?

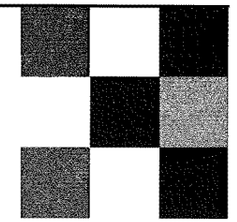
29 Create the following file, and then give the command *make*. What gets printed? Write your answer first; then experiment to check it.

In file makefile

```
# Experiment with the make command!
Start: A1 A2
      @echo Start
A1: A3
      @echo A1
A2: A3
      @echo A2
A3:
      @echo A3
```


30 What gets printed?

```
#include <stdio.h>

int main(void)
{
   printf("Hello!\n");
   fclose(stdout);
   printf("Goodbye!\n");
   return 0;
}
```



Chapter 12

Advanced Applications

C has the ability to be close to the machine. It was originally used to implement UNIX, and building systems is still one of its important uses. In this chapter, we describe some advanced applications, including material useful for systems programmers. We also discuss the use of matrices for engineers and scientists.

12.1 Creating a Concurrent Process with fork()

UNIX is a multiuser, multiprocessing operating system. Each process has a unique process identification number. The following command will show you what your machine is currently doing:

```
ps -aux
```

Here is an example of its output:

```
USER       PID %CPU %MEM    SZ  RSS TT STAT START  TIME COMMAND
blufox   17725 34.0  1.6   146  105 i2 R    15:13   0:00 ps -aux
amber    17662  1.4  7.0   636  469 j5 S    15:10   0:08 vi find.c
root       143  0.5  0.1     5    3 ?  S    Jul 31 10:17 /etc/update
.....
```

The first line contains headings. The remaining lines supply information about each process: the user's login name, the process identification number, and so forth. (Read the online manual to find out more about the *ps* command.) The operating system runs many processes concurrently by time-sharing the machine resources.

In UNIX, the programmer can use `fork()` to create a new process, called the *child process*, that runs concurrently with the *parent process*. (It is not part of ANSI C.) The function `fork()` takes no arguments and returns an `int`. The following is a simple program illustrating its use:

```
#include <stdio.h>

int main(void)
{
   int   fork(void), value;

   value = fork();                              /* new process */
   printf("In main: value = %d\n", value);
   return 0;
}
```

The output of this program changes each time we run it. Here is example output:

```
In main: value = 17219
In main: value = 0
```

As noted, when `fork()` is invoked, it creates a new process, the child process. This new process is an exact copy of the calling process, except that it has its own process identification number. The function call `fork()` returns 0 to the child, and it returns the child's process ID to the parent. In the output of our program, the first line was printed by the parent, the second line by the child.

Let us modify this program by adding to the code a second copy of the statement

```
value = fork();
```

Here is what gets printed:

```
In main: value = 17394
In main: value = 0
In main: value = 0
In main: value = 17395
```

The children have unique process identification numbers. The program created four concurrent versions of `main()`. The order of execution of these processes is system-dependent and *nondeterministic*—that is, the order is not necessarily the same for each execution of the program. (See exercise 1, on page 586.) *Caution:* Invoking `fork()` too many times can cause the system to fail by exhausting all available processes.

When `fork()` is invoked, it creates two processes, each with its own set of variables. If a file pointer is used, however, care must be taken because the file pointer in each of the processes will refer to the same underlying file.

The value returned by `fork()` can be used in an `if-else` statement to discriminate between the actions of the child and the parent. In our next program, we compute Fibonacci numbers in the child process and print elapsed time in the parent process. We use `sleep()` to suspend execution of the parent process for 2-second intervals.

```
/* Compute Fibonacci numbers and print time asynchronously. */

#include <stdio.h>
#include <time.h>

int    fib(int);
int    fork(void);
void   sleep(unsigned);

int main(void)
{
   int   begin = time(NULL), i;

   if (fork() == 0)                                   /* child */
      for (i = 0; i < 30; ++i)
       printf("fib(%2d) = %d\n", i, fib(i));
   else                                               /* parent */
      for (i = 0; i < 30; ++i) {
       sleep(2);
       printf("elapsed time = %d\n", time(NULL) - begin);
      }
   return 0;
}

int fib(int n)
{
   if (n <= 1)
      return n;
   else
      return (fib(n - 1) + fib(n - 2));
}
```

When this program is executed, the outputs of the two processes are intermixed in a nondeterministic fashion.

12.2 Overlaying a Process: the exec . . . () Family

From within a program, the current process, meaning the program itself, can be overlaid with another process. To do this, the programmer calls a member of the `exec...()` family. (See Section A.16, "Miscellaneous," on page 682, for a list of all members.) These functions are not part of ANSI C, but, typically, they are available in both MS-DOS and UNIX.

We want to use the `execl()` function to illustrate how one process can be overlaid with another. The function prototype is

```
int   execl(char *path, char *arg0, ...);
```

In MS-DOS, this prototype is provided typically in the header file *process.h*. In UNIX, it can be in this header file or in some other. The first argument is the path of the executable file—that is, the new process. The remaining arguments correspond to the command line arguments expected by the new process. The argument list ends with the null pointer 0. The value -1 is returned if the executable file cannot be found or is nonexecutable.

Before we use `execl()` in a program, let us write two other small programs. We will use the compiled code to overlay another process.

In file pgm1.c

```
#include <stdio.h>

int main(int argc, char **argv)
{
   int   i;

   printf("%s: ", argv[0]);
   for (i = 1; i < argc; ++i)      /* print the arg list */
      printf("%s ", argv[i]);
   putchar('\n');
   return 0;
}
```


In file pgm2.c

```
#include <stdio.h>

int main(int argc, char **argv)
{
   int   i, sum = 0, value;

   for (i = 0; i < argc; ++i)      /* sum the arguments */
      if (sscanf(argv[i], "%d", &value) == 1)
         sum += value;
   printf("%s: sum of command line args = %d\n", argv[0], sum);
   return 0;
}
```

Next, we compile the two programs:

```
cc -o pgm1 pgm1.c;   cc -o pgm2 pgm2.c
```

Observe that *pgm1* and *pgm2* are executable files in the current directory. In our next program, we will overlay the parent process with one of these two processes.

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

int main(void)
{
   int   choice = 0;

   printf("%s\n%s\n%s",
      "The parent process will be overlaid.",
      "You have a choice.",
      "Input 1 or 2: ");
   scanf("%d", &choice);
   putchar('\n');
   if (choice == 1)
      execl("pgm1", "pgm1", "a", "b", "c", 0);
   if (choice == 2)
      execl("pgm2", "pgm2", "1", "2", "3", "go", 0);
   printf("ERROR: You did not input 1 or 2.\n");
   return 0;
}
```

If we run this program under MS-DOS and enter 1 when prompted, here is what appears on the screen:

```
The parent process will be overlaid.
You have a choice.
Input 1 or 2: 1

C:\CENTER\PGM1.EXE: a b c
```

When a process is successfully overlaid, there is no return to the parent. The new process takes over completely.

In UNIX, `fork()` is often used when overlaying one process with another.

```
if (fork() == 0)
   execl("/c/c/bf/bin/mmf", "mmf", "-f", 0);
                  /* execute mmf */
else
   .....          /* do something else */
```

Using the spawn...() Family

Most C systems in MS-DOS provide the `spawn...()` family of functions. This family is similar to the `exec...()` family, except that the first argument is an integer mode.

Modes for the spawn...() family	Meaning
0	Parent process waits until child process completes execution.
1	Concurrent execution—not yet implemented.
2	Child process overlays the parent process; same as equivalent `exec...()` call.

MS-DOS is not a multiprocessing operating system. Where a programmer might use `fork()` and `execl()` in UNIX, `spawnl(0, ...)` could be used instead in MS-DOS. Here is a call to `spawnl()` that invokes *chkdsk*:

```
spawnl(0, "c:\chkdsk", "chkdsk", "c:", "/f", 0);
```

Because the mode is 0, the parent process will wait until the child process *c:\chkdsk* completes execution before continuing with its work.

12.3 Interprocess Communication Using pipe()

In UNIX, the programmer can use `pipe()` to communicate between concurrent processes. The function prototype is given by

```
int   pipe(int pd[2]); /* pd stands for pipe descriptor. */
```

The function call `pipe(pd)` creates an input/output mechanism called a pipe. Associated file descriptors, or pipe descriptors, are assigned to the array elements `pd[0]` and `pd[1]`. The function call returns 0 if the pipe is created, and returns -1 if there is an error.

After a pipe has been created, the system assumes that two or more cooperating processes created by subsequent calls to `fork()` will use `read()` and `write()` to pass data through the pipe. One descriptor, `pd[0]`, is read from, and the other, `pd[1]`, is written to. The pipe capacity is implementation-dependent, but is at least 4,096 bytes. If a write fills the pipe, the pipe is blocked until data are read out of it. As with other file descriptors, `close()` can be used to explicitly close `pd[0]` and `pd[1]`.

To illustrate the use of `pipe()`, we will write a program that computes the sum of the elements of an array. We compute the sum of each row concurrently in a child process and write the values on a pipe. In the parent process, we read the values from the pipe.

In file concurrent_sum.c

```
/*  Use pipes to sum N rows concurrently. */

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

#define   N   3

int    add_vector(int v[]);
void   error_exit(char *s);
int    fork(void);
int    pipe(int pd[2]);
int    read(int fd, void *buf, unsigned len);
int    write(int fd, void *buf, unsigned len);
```

```
int main(void)
{
   int   a[N][N] = {{1, 1, 1}, {2, 2, 2}, {3, 3, 3}},
         i, row_sum, sum = 0,
         pd[2];                                  /* pipe descriptors */

   if (pipe(pd) == -1)                           /* create a pipe */
      error_exit("pipe() failed");
   for (i = 0; i < N; ++i)
      if (fork() == 0) {                         /* child process */
         row_sum = add_vector(a[i]);
         if (write(pd[1], &row_sum, sizeof(int)) == -1)
            error_exit("write() failed");
         return;                                 /* return from child */
      }
   for (i = 0; i < N; ++i) {
      if (read(pd[0], &row_sum, sizeof(int)) == -1)
         error_exit("read() failed");
      sum += row_sum;
   }
   printf("Sum of the array = %d\n", sum);
   return 0;
}

int add_vector(int v[])
{
   int i, vector_sum = 0;

   for (i = 0; i < N; ++i)
      vector_sum += v[i];
   return vector_sum;
}

void error_exit(char *s)
{
   fprintf(stderr, "\nERROR: %s - bye!\n", s);
   exit(1);
}
```


Dissection of the *concurrent_sum* Program

```
■  if (pipe(pd) == -1)                           /* create a pipe */
      error_exit("pipe() failed");
```

A pipe is created before other processes are forked. If the call to `pipe()` fails, we invoke `error_exit()` to write a message to the user and exit the program.

```
■  for (i = 0; i < N; ++i)
      if (fork() == 0) {                         /* child process */
         row_sum = add_vector(a[i]);
         if (write(pd[1], &row_sum, sizeof(int)) == -1)
            error_exit("write() failed");
         return;                                 /* return from child */
      }
```

Each time through the loop, we use `fork()` to create a child process. After `row_sum` has been computed, we write it on the pipe with the function call

```
write(pd[1], &row_sum, sizeof(int))
```

If the call to `write()` fails, we invoke `error_exit()` to write a message to the user and exit the program. Note carefully that we explicitly `return` from the child after the call to `write()`. If we do not do this, then the children will themselves create children each time through the loop.

```
■  for (i = 0; i < N; ++i) {
      if (read(pd[0], &row_sum, sizeof(int)) == -1)
         error_exit("read() failed");
      sum += row_sum;
   }
```

In the parent process, we invoke `read()` to read `row_sum` from the pipe. If the call to `read()` fails, we invoke `error_exit()` to write a message to the user and exit.

12.4 Signals

An exceptional condition, or signal, is generated by an abnormal event. For example, the user may type a control-c to effect an interrupt, or a program error may cause a bus error or a segmentation fault. A floating-point exception occurs when two very large floating numbers are multiplied, or when division by zero is attempted. ANSI C provides the function `signal()` in the standard library. Its function prototype and some macros are in *signal.h*. The material in this section applies to both MS-DOS and UNIX.

The exceptional conditions that can be handled by the operating system are defined as symbolic constants in *signal.h*. Some examples are

```
#define   SIGINT    2      /* interrupt */
#define   SIGILL    4      /* illegal instruction */
#define   SIGFPE    8      /* floating-point exception */
#define   SIGSEGV   11     /* segment violation */
```

Although the signals that can be handled are system-dependent, the ones that we have listed are common to most C systems.

If an exceptional condition is raised in a process, then the typical default action of the operating system is to terminate the process. The programmer can use `signal()` to invoke a signal handler that replaces the default system action. The function prototype is

```
void   (*signal(int sig, void (*func)(int)))(int);
```

This function takes two arguments, an `int` and a pointer to a function that takes an `int`, and returns nothing. The function returns a pointer to a function that takes an `int` and returns nothing.

The function call `signal(sig, func)` associates the signal `sig` with the signal handler `func()`. This causes the system to pass `sig` as an argument to `func()` and invoke it when the signal `sig` is raised.

Some special signal handlers are defined as macros in *signal.h*. We will use two of them:

```
#define   SIG_DFL   ((void (*)(int)) 0)      /* default */
#define   SIG_IGN   ((void (*)(int)) 1)      /* ignore */
```

The casts cause the constants to have a type that matches the second argument to `signal()`. Here is an example of how the second macro gets used:


```
signal(SIGFPE, SIG_IGN);      /* ignore floating-point exceptions */
```

This causes floating-point exceptions to be ignored by the system. If at some later point we want to resume the default action, we can write

```
signal(SIGFPE, SIG_DFL);      /* take default action */
```

If a standard signal handler is inappropriate in a given application, the programmer can use `signal()` to catch a signal and handle it as desired. Imagine wanting to use an interrupt to get the attention of a program without terminating it. The following program illustrates this:

```
/* Using a signal handler to catch a control-c. */

#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

#define   MAXSTRING   100

void   cntrl_c_handler(int sig);
int    fib(int n);

int main(void)
{
   int   i;

   signal(SIGINT, cntrl_c_handler);
   for (i = 0; i < 46; ++i)
      printf("fib(%2d) = %d\n", i, fib(i));
   return 0;
}

void cntrl_c_handler(int sig)
{
   char   answer[MAXSTRING];

   printf("\n\n%s%d\n\n%s",
      "Interrupt received!  Signal = ", sig,
      "Do you wish to continue or quit? ");
   scanf("%s", answer);
   if (*answer == 'c')
      signal(SIGINT, cntrl_c_handler);
   else
      exit(1);
}
```

The function `fib()` is not shown. It is the same function that we used in Section 12.1, "Creating a Concurrent Process with `fork()`," on page 557.

Dissection of the *fib_signal* Program

- `signal(SIGINT, cntrl_c_handler);`

If the `SIGINT` signal is raised, the system catches it and passes control to the function `cntrl_c_handler()`.

- ```
  void cntrl_c_handler(int sig)
  {
     char   answer[MAXSTRING];
     printf("\n\n%s%d\n\n%s",
        "Interrupt received!  Signal = ", sig,
        "Do you wish to continue or quit? ");
  ```

When `signal()` passes control to this function, a message is printed on the screen.

- ```
     if (*answer == 'c')
        signal(SIGINT, cntrl_c_handler);
     else
        exit(1);
  ```

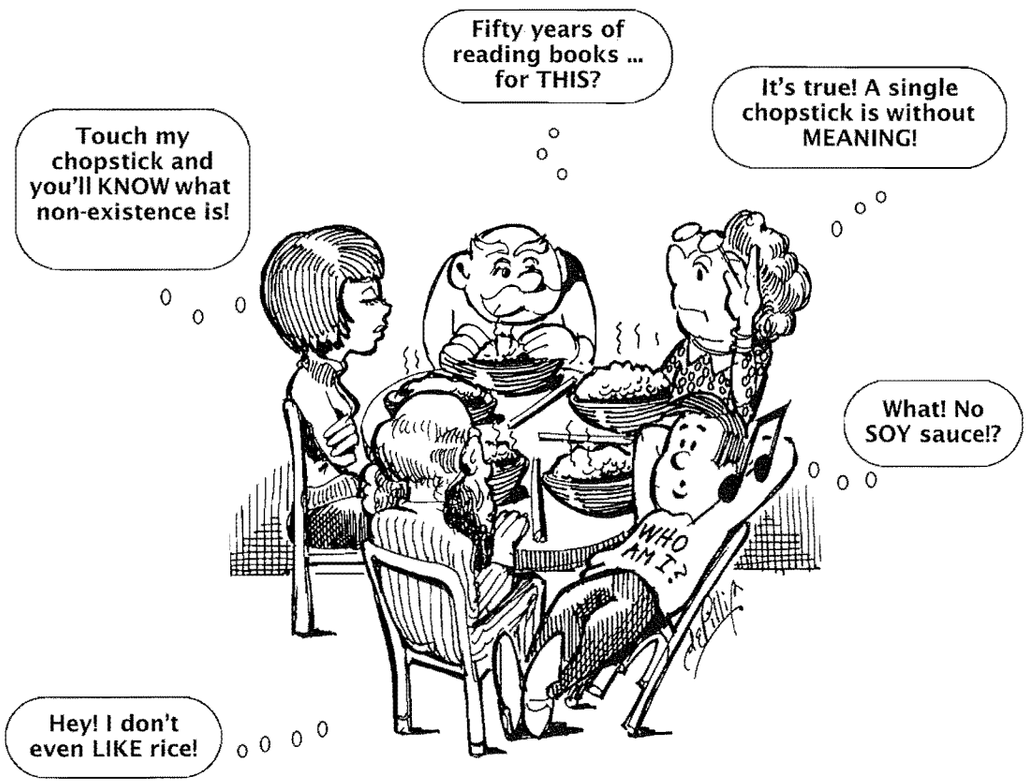
Depending on the answer typed by the user, we either reset our signal handling mechanism or we exit the program. The interrupt signal is special. On some systems, after an interrupt has occurred, the system reverts to default action. The statement

```
signal(SIGINT, cntrl_c_handler);
```

allows us to catch another interrupt.

12.5 An Example: The Dining Philosophers

The dining philosophers problem is a standard model for synchronizing concurrent processes that share resources. Five philosophers are seated around a circular table. Each philosopher has one chopstick at his side and a bowl of rice in front of him. Eating rice requires two chopsticks. There are only five chopsticks, so at most two philosophers can be eating at any one time. Indeed, if each philosopher picked up one chopstick, none could have two and they would all be *deadlocked*. A philosopher may acquire only the chopsticks to his or her immediate left or right. The problem is to write a program with concurrent processes representing the philosophers, where each philosopher gets to eat fairly often.



In our program, each philosopher will be an identically forked process. The solution will require the use of semaphores implemented as pipes. A *semaphore* is a special variable allowing *wait* and *signal* operations. (The `signal()` function that we use here has nothing to do with `signal()` in the standard library, which we discussed in the previous section.) The variable is a special location for storing unspecified values. The wait operation expects one item and removes it. The signal operation adds one item. The wait operation blocks a process until it can accomplish its removal operation. The signal operation can start up a blocked process.

In file dining.c

```
/* The dining philosopher program. */

#include <stdio.h>
#include <stdlib.h>                 /* for calloc() and exit() */

#define   N                 5       /* number of philosophers */
#define   Busy_Eating       1
#define   Busy_Thinking     1
#define   Left(p)           (p)           /* chopstick macros */
#define   Right(p)          (((p) + 1) % N)

typedef     int *   semaphore;

semaphore   chopstick[N];                       /* global array */
int         fork(void);
semaphore   make_semaphore(void);
void        philosopher(int me);
void        pick_up(int me);
int         pipe(int pd[2]);
void        put_down(int me);
int         read(int fd, void *buf, unsigned len);
void        signal(semaphore s);
void        sleep(unsigned seconds);
void        wait(semaphore s);
int         write(int fd, void *buf, unsigned len);
```



```
int main(void)
{
   int   i;

   for (i = 0; i < N; ++i) {     /* put chopsticks on the table */
      chopstick[i] = make_semaphore();
      signal(chopstick[i]);
   }
   for (i = 0; i < N - 1; ++i)   /* create philosophers */
      if (fork() == 0)
       break;
   philosopher(i);               /* all executing concurrently */
   return 0;
}
```

The function `main()` creates each chopstick as a semaphore. Each chopstick starts out as an available resource. Then each philosopher is created as a concurrent forked process. Each philosopher executes `philosopher(i)`, the routine that attempts to alternately eat and think.

```
/* Acquire chopsticks, input is philosopher number. */

void pick_up(int me)
{
   if (me == 0) {
      wait(chopstick[Right(me)]);
      printf("Philosopher %d picks up right chopstick\n", me);
      sleep(1);  /* simulate slow picking encourage deadlock */
      wait(chopstick[Left(me)]);
      printf("Philosopher %d picks up left chopstick\n", me);
   }
   else {
      wait(chopstick[Left(me)]);
      printf("Philosopher %d picks up left chopstick\n", me);
      sleep(1);  /* simulate slow pick up encourage deadlock */
      wait(chopstick[Right(me)]);
      printf("Philosopher %d picks up right chopstick\n", me);
   }
}

/* Relinquish chopsticks, input is the philosopher number. */

void put_down(int me)
{
   signal(chopstick[Left(me)]);
   signal(chopstick[Right(me)]);
}
```

```
/* Philosopher process, input is the philosopher number. */

void philosopher(int me)
{
   char   *s;
   int    i = 1;

   for ( ; ; ++i) {     /* forever */
      pick_up(me);
      s = i == 1 ? "st" : i == 2 ? "nd" : i == 3 ? "rd" : "th";
      printf("Philosopher %d eating for the %d%s time\n",
            me, i, s);
      sleep(Busy_Eating);
      put_down(me);
      printf("Philosopher %d thinking\n", me);
      sleep(Busy_Thinking);
   }
}
```

The `philosopher()` routine attempts to acquire a left and right chopstick. If successful, it eats and returns the chopsticks and resumes thinking. It acquires the chopsticks by using the semaphore operation `wait()` on its left and right chopsticks. The function `pick_up()` is blocked until both chopsticks are acquired. It releases the chopsticks by using the semaphore operation `signal()` on its left and right chopsticks. The function `put_down()` is terminated when both chopsticks are released.

```
semaphore make_semaphore(void)
{
   int   *sema;

   sema = calloc(2, sizeof(int));     /* permanent storage */
   pipe(sema);
   return sema;
}

void wait(semaphore s)
{
   int   junk;

   if (read(s[0], &junk, 1) <= 0) {
      printf("ERROR: wait() failed, check semaphore creation.\n");
      exit(1);
   }
}
```



```
void signal(semaphore s)
{
   if (write(s[1], "x", 1) <= 0) {
      printf("ERROR: write() failed, check semaphore creation.\n");
      exit(1);
   }
}
```

The semaphore is constructed by a call to `pipe(sema)`. The semaphore function `wait()` is blocked until an item of input can be read into `junk`. The semaphore function `signal()` produces an item of output.

This example is a standard one for operating system resource allocation in a multiprocessing environment. A detailed discussion of algorithms for these problems can be found in *The Logical Design of Operating Systems* by Lubomir Bic and Alan Shaw (Englewood Cliffs, N.J.: Prentice-Hall, 1988).

12.6 Dynamic Allocation of Matrices

Engineers and scientists use matrices extensively. In this section, we explain how a matrix can be created dynamically as an array of pointers so that it can be passed to functions that are designed to work on matrices of different sizes.

Why Arrays of Arrays Are Inadequate

In Section 6.12, "Multidimensional Arrays," on page 277, we discussed the simplest way of implementing matrices. Let us briefly review how this is done, and explain why this is unacceptable for lots of applications. If we want a 3×3 matrix, for example, we can declare

```
double   a[3][3];
```

This allocates space for a as an array of arrays. If we want to work with a locally, there is no problem. However, lots of operations on matrices, such as finding determinants or computing eigen values, are best done by calling a function. For the purpose of discussion, suppose we want to find the determinant of a. After the matrix a has been filled, we want to be able to write something like

```
det = determinant(a);
```

The function definition for `determinant()` will look like

```
double determinant(double a[][3])
{
   .....
```

Because the compiler needs the 3 to build the correct storage mapping function, our determinant function can be used only on 3×3 matrices. If we want to compute the determinant of a 4×4 matrix, we need to write a new function definition. This is unacceptable. We want to be able to write a determinant function that can be used for square matrices of any size.

Building Matrices with Arrays of Pointers

By starting with the type pointer to pointer to `double`, we can build a matrix of whatever size we want, and we can pass it to functions that are designed to work on matrices of any size. Let us start with the code we need to create the space for a matrix:

```
int      i, j, n;
double   **a, det, tr;

.....                                  /* get n from somewhere */

a = calloc(n, sizeof(double *));
for (i = 0; i < n; ++i)
   a[i] = calloc(n, sizeof(double));
```

The size of the matrix does not have to be known *a priori*. We can get `n` from the user or read it from a file or compute it. It does not have to be coded as a constant in the program. Once we know the desired size, we use the standard library function `calloc()` to create the space for the matrix dynamically. The function prototype is given in *stdlib.h* as

```
void   *calloc(size_t nitems, size_t size);
```

Typically, the type definition for `size_t` is given by

```
typedef   unsigned   size_t
```

This type definition can be found in *stddef.h* and sometimes in *stdlib.h*, too. If `size_t` is not defined directly in *stdlib.h*, then *stddef.h* will be included there. A function call of the form

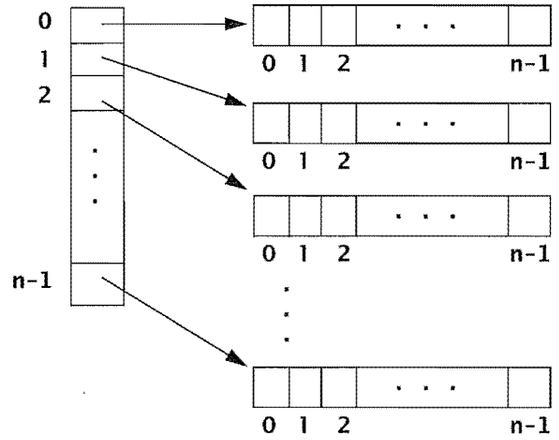

```
calloc(n, sizeof( type ))
```

allocates space in memory for an array of `n` elements of the specified type. The base address of the array is returned. Thus, the statement

```
a = calloc(n, sizeof(double *));
```

allocates space for `a`, which we can now think of as an array of pointers to `double` of size `n`. In the `for` loop, each element of `a` is assigned the base address of space allocated in memory for an array of `double`s of size `n`. We can think of `a` in memory as

An n x n matrix in memory



Now that space has been created for the matrix, we want to fill it. Let us assign to the elements of a integer values that are randomly distributed in the range from -9 to +9.

```
for (i = 0; i < n; ++i)
   for (j = 0; j < n; ++j)
      a[i][j] = rand() % 19 - 9;      /* from -9 to +9 */
```

Note that even though our matrix `a` is stored in memory as an array of pointers, the usual matrix expression `a[i][j]` is used to access the element in the *i*th row, *j*th column (counting from zero). Because `a` is of type `double **`, it follows that `a[i]` is of type `double *`. It can be thought of as the *i*th row (counting from zero) of the matrix. Because `a[i]` is of type `double *`, it follows that `a[i][j]` is of type `double`. It is the *j*th element (counting from zero) in the *i*th row.

Now that we have assigned values to the elements of the matrix, we can pass the matrix as an argument to various functions. Suppose we want to print it, compute its determinant, and compute its trace. In the calling environment we can write

```
print_matrix(a, n);
det = determinant(a, n);
tr = trace(a, n);
```

The function definition for `print_matrix()` is given by

```
void print_matrix(double **a, int n)
{
   int   i, j;

   for (i = 0; i < n; ++i) {
      for (j = 0; j < n; ++j)
         printf("%7.1f", a[i][j]);
      putchar('\n');
   }
   putchar('\n');
}
```

The function definition for `determinant()` begins as

```
double determinant(double **a, int n)
{
   .....
```

Because the determinant function is complicated, we postpone further discussion until the exercises. The trace, however, is easy. By definition, the trace of a matrix is the sum of its diagonal elements:

```
double trace(double **a, int n)
{
   int      i;
   double   sum = 0.0;

   for (i = 0; i < n; ++i)
      sum += a[i][i];
   return sum;
}
```

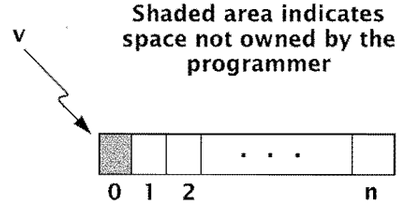

Adjusting the Subscript Range

In mathematics, the subscripts for vectors and matrices usually start at 1, not 0. We can arrange for our C code to do this, too. The following function can be used to create space for an *n*-vector:

```
double *get_vector_space(int n)
{
   int      i;
   double   *v;

   v = calloc(n, sizeof(double));
   return (v - 1);                    /* offset the pointer */
}
```

Because the pointer value that is returned has been offset to the left, subscripts in the calling environment will run from 1 to *n* rather than from 0 to *n* - 1.

An n-vector indexed from 1, not 0



Note that in the picture, we have shaded an element to indicate that the programmer does not own element 0. Now we can write code like the following:

```
int        n;
double     *v;

.....                                 /* get n from somewhere */
v = get_vector_space(n);
for (i = 1; i <= n; ++i)
   v[i] = rand() % 19 - 9;            /* from -9 to +9 */
.....
```

Instead of offsetting a pointer, we always have the option of allocating more space and then not using it all.

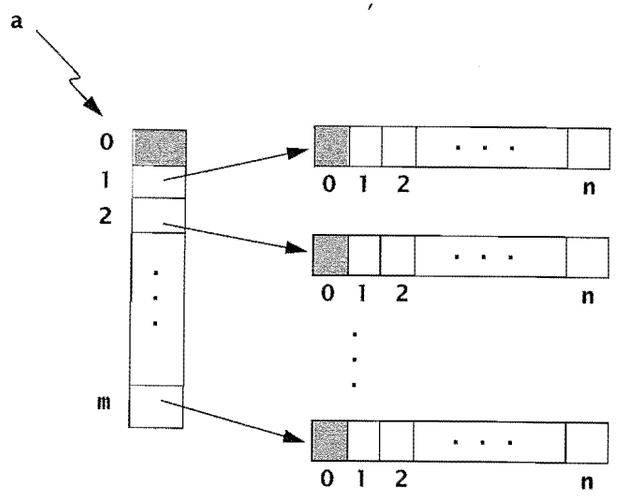
```
v = calloc(n + 1, sizeof(double));      /* allocate extra space */
for (i = 1; i <= n; ++i)
   v[i] = ...
```

The technique of offsetting the pointer is better in the sense that no space is wasted. Let us use the technique to write a function that creates space for an m × n matrix with subscripts that start at 1 rather than 0.

```
double **get_matrix_space(int m, int n)
{
   int      i;
   double   **a;

   a = calloc(m, sizeof(double *));
   --a;                                  /* offset the pointer */
   for (i = 1; i <= m; ++i) {
      a[i] = calloc(n, sizeof(double));
      --a[i];                            /* offset the pointer */
   }
   return a;
}
```

We can think of the matrix a in memory as

An m x n matrix indexed from 1, not 0



Note that once again we have used shaded areas to indicate space not owned by the programmer.

If a programmer wants space previously allocated by `calloc()` to be made available to the system again, then the function `free()` must be used. Let us suppose that in `main()` we write

```
int      i, j, n;
double   **a;

for ( ; ; ) {                                  /* do it forever */
   .....                                /* get n from somewhere */
   a = get_matrix_space(n, n);
   for (i = 1; i <= m; ++i)
      for (j = 1; j <= n; ++j)
         a[i][j] = ...                         /* assign values */
   .....                                        /* do something */
   release_matrix_space(a, m);
}
```

Here is the function definition for `release_matrix_space()`. We have to be careful to undo the pointer offsets that occurred in `get_matrix_space()`.

```
void release_matrix_space(double **a, int m)
{
   int   i;

   for (i = 1; i <= m; ++i)
      free(a[i] + 1);
   free(a + 1);
}
```

Allocating All the Memory at Once

In certain applications, especially if the matrices are large, it may be important to allocate all the matrix space at once. At the same time, we want our matrices to be indexed from 1, not 0. Here is code that we can use to allocate space for our matrices:

In file matrix.h

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

typedef   double **   matrix;
typedef   double *    row;
typedef   double      elem;
```

```
.....
matrix   get_matrix_space(int m, int n);
void     release_matrix_space(matrix a);
void     fill_matrix(matrix a, int m, int n);
void     prn_matrix(const char *s, matrix a, int m, int n);
```

Note that we can get different kinds of matrices just by changing the typedef appropriately.

In file space.c

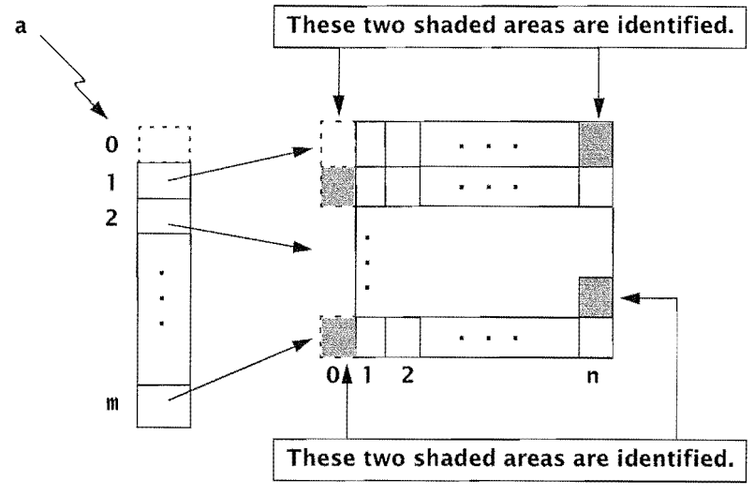
```
#include "matrix.h"

matrix get_matrix_space(int m, int n)
{
   int      i;
   elem *   p;
   matrix   a;

   p = malloc(m * n * sizeof(elem));   /* get space all at once */
   a = malloc(m * sizeof(row));
   --a;                                /* offset the pointer */
   for (i = 1; i <= m; ++i)
      a[i] = p + ((i - 1) * n) - 1;
   return a;
}

void release_matrix_space(matrix a)
{
   elem *   p;

   p = (elem *) a[1] + 1;          /* base address of the array */
   free(p);
   free(a + 1);
}
```

Suppose that in some other function we write

```
int      m, n;
matrix   a;

.....                              /* get m and n from somewhere */
a = get_matrix_space(m, n);
```


Here is how we can think of a in memory:

Space for an m x n matrix allocated all at once and indexed from 1, not 0



Caution: If we use algorithms that swap rows and we want to deallocate our matrix space, then we have to be careful not to lose the base address of the array in memory. We may want to redesign our abstract data type (ADT) matrix.

12.7 Returning the Status

Throughout this text, at the end of every function definition for `main()` we have written the line

```
return 0;
```

In this section, we want to explain how the value returned from `main()` can be used by the operating system. From the viewpoint of the operating system, the value returned by a process is called its *status*. Processes include both programs and shell scripts. The operating system does not necessarily have to use the status that gets returned from a process.

In most operating systems, you communicate with the machine by clicking the mouse or typing commands on the keyboard. When you give commands such as

cc pgm.c date echo Beautiful!

you are communicating via the shell. The shell (which is itself a program) interprets what you write and tries to execute your commands. There are three shells commonly found on UNIX systems: the Bourne shell *sh*, the C shell *csh*, and the Korn shell *ksh*.

To show how the value returned from `main()` (the program status) can be used, we are going to write a small program and then write a shell script that invokes our program. Here is our program:

In file try_me.c

```
#include <stdio.h>

#define   MARKER   ">> "

int main(int argc, char **argv)
{
   int   val;

   printf("\n");
   printf(MARKER "    Input an integer: ");
   scanf("%d", &val);
   printf(MARKER "Value being returned: %d\n\n", val);
   return val;
}
```

Note that `main()` can return any integer value, not just zero. Note also that we are using a marker to clearly indicate the lines that are written to the screen.

Next, we compile this program, putting the executable file in a directory that is in our path.

gcc -o $base/bin/try_me try_me.c

Now we are ready to write our shell script.

In file go_try

```
#!/usr/bin/csh

##
## Experiment with the status.
##
```



```
echo ---
echo At the top: status = $status
try_me
echo ---
echo After try_me: status = $status
try_me
set val = $status
echo ---
echo After try_me again: val = $val
echo ""
echo ---
echo To exit from the while loop, input 0.
while ($val)
    try_me
    set val = $status
    echo In the loop: val = $val
end
```

Dissection of the *go_try* Shell Script

■ `#!/usr/bin/csh`

A comment starts with a # and continues to the end of the line. This line is special, however, because it begins with `#!` in column 1 at the top of the file. Lines such as this are read by the operating system. This line tells the operating system to use the C shell, which is */usr/bin/csh*, to execute this file.

■ `## Experiment with the status.`

Comments begin with a # and continue to the end of the line.

■
```
echo ---
echo At the top: status = $status
```

The `echo` command takes the remainder of the line and writes it to the screen. The symbol $, however, is special. The value of a shell variable is obtained by writing

$(*shell_variable_name*)

The shell variable `status` is a built-in variable.

```
try_me
echo ---
echo After try_me: status = $status
```

At this point, we invoke our program *try_me*. After we have done this, we want to examine the program status. (As we will see, our attempt fails.)

```
try_me
set val = $status
echo ---
echo After try_me again: val = $val
```

After we invoke *try_me* again, we use the *set* command to create a shell variable named `val` that has, as its value, the current value of `$status`. (As we shall see, `val` has the value returned by `main()` in the program *try_me*.) We use the *echo* command to display `$val` on the screen.

```
echo To exit from the while loop, input 0.
while ($val)
   try_me
   set val = $status
   echo In the loop: val = $val
end
```

Finally, we enter a loop so that we can experiment and see the results printed on the screen.

Our shell script *go_try* is not very informative by itself. When we execute it, however, a lot of new ideas start to fall into place. To make the file executable, we give the command

chmod u+x go_try

Finally, we give the command

go_try

and enter some numbers when prompted. Here is what appears on the screen:


```
---
At the top: status = 0

>>     Input an integer: 3
>> Value being returned: 3

---
After try_me: status = 0

>>     Input an integer: 7
>> Value being returned: 7

---
After try_me again: val = 7

.....
```

Dissection of the Output From *go_try*

```
---
At the top: status = 0
```

The lines that begin with >> were written by the *try_me* program. The nonblank lines were written by *echo* commands in the shell script *go_try*. Every process resets the value of `status` upon exiting, and by convention 0 is used to signify normal termination of a process. An *echo* command is itself a process. The very first one we did was

```
echo ---
```

When this process terminated, it set the value of `status` to 0. This is reflected in the output line

```
At the top: status = 0
```

```
>>     Input an integer: 3
>> Value being returned: 3
```

These are lines generated by the *try_me* program. When prompted, we typed a 3 at the keyboard. The second line reports this.

```
---
After try_me: status = 0
```

At first, this report surprised us, but then we remembered that `status` is ephemeral. The *echo* command that generated --- reset the `status` after *try_me*. Hence, the value that gets reported is 0, not 3.

```
>>     Input an integer: 7
>> Value being returned: 7

---
After try_me again: val = 7
```

After *try_me* is invoked again, we enter 7 when prompted. Immediately after *try_me* exits, we use the shell variable `val` to capture the current value of `$status`. In the shell script, the relevant lines are

```
try_me
set val = $status
echo ---
echo After try_me again: val = $val
```

The `while` loop in the shell script *go_try* allows the user to experiment. We observe that when we enter an integer n, the value that gets printed is n mod 256. In particular, -1 yields 255. Thus, the program status is a nonnegative number between 0 and 255. When `main()` returns a value to the shell, the shell can use it for its own purposes, if this is desirable.

Although we used the UNIX C shell in our discussion in this section, the story is similar with respect to the other UNIX shells and with respect to MS-DOS.

Summary

1 A concurrent process is code that executes simultaneously with the code that invoked it. In UNIX, `fork()` can be used to create a child process that is a copy of the parent process, except that the child has its own unique process identification number. If a call to `fork()` is successful, it returns 0 to the child and the child's process identification number to the parent; if unsuccessful, it returns -1.

2 In both MS-DOS and UNIX, a member of the `exec...()` family can be used to overlay one process with another. There is no return to the parent. Another family, `spawn...()`, is available in MS-DOS. With this family, it is possible to return to the parent. This use, in a certain sense, is comparable to the combined use of `fork()` and `exec...()` in UNIX.

3 In UNIX, the system call `pipe(pd)` creates a mechanism for interprocess communications called a pipe. After a pipe is open, calls to `fork()` are used to create processes that communicate via the pipe. The functions `read()` and `write()` are used.

4 The function `signal()` is available in the standard library. It can be used to associate a signal with a signal handler. The signal handler can be a function the programmer writes to replace the default system action. When the signal is raised, program control passes to the signal handler. The set of signals that are handled by the operating system is defined as macros in *signal.h*. This set is system-dependent, but some of the signals are common to both MS-DOS and UNIX.

5 The dining philosophers problem is a standard model for synchronizing concurrent processes that share resources. The problem is to write a program with concurrent processes representing the philosophers, where each philosopher gets to eat (share resources) fairly often.

6 A *semaphore* is a special variable allowing *wait* and *signal* operations. The variable is a special location for storing unspecified values. The wait operation expects one item and removes it. The signal operation adds one item. The wait operation blocks a process until it can accomplish its removal operation. The signal operation can start up a blocked process.

7 By starting with the type pointer to pointer to `double`, we can build a matrix of whatever size we want, and we can pass it to functions that are designed to work on matrices of any size. This is an important idea for engineers and scientists.

Exercises

1 Modify the simple forking program given in Section 12.1, "Creating a Concurrent Process with `fork()`," on page 556, so that it has three copies of the line

```
value = fork();
```

The output of the program is nondeterministic. Explain what this means. *Hint:* Execute your program repeatedly.

2 If `fork()` fails, no child process is created and -1 is returned. Write a program that contains the following code:

```
#define   N   3

for (i = 1; i <= N; ++i) {
   pid = fork();
   if (pid == 0)
      printf("%2d: Hello from child\n", i);
   else if (pid > 0)
      printf("%2d: Hello from parent\n", i);
   else
      printf("%2d: ERROR: Fork did not occur\n", i);
}
```

How large must `N` be on your system to get an error message printed?

3 In Section 12.2, "Overlaying a Process: the `exec...()` Family," on page 560, we presented a program that illustrates how one process gets overlaid with another. Modify that program. Begin by creating another executable file, say *pgm3*, which prints the current date. Provide that as another choice for overlaying the parent process in the program you are modifying.

4 Does the *fortune* command work on your system? If so, find out where the executable code is. Then take one of your working programs and insert code similar to

```
if (fork() == 0)
   execl("/usr/games/fortune", "fortune", 0);
```

What is the effect of doing this?

5 Write a program that uses n concurrent processes to multiply two n × n matrices.

6 Compile two programs into executables named *prog1* and *prog2*. Write a program that executes both concurrently by forking twice and using `execl()` to overlay the two executables.

7 When a pipe is filled to capacity on your system, how many characters does it hold? Write a program to find out. *Hint:* Write characters into the pipe until it blocks.

8 Change the dining philosophers program so that upon receiving an interrupt it prints out how many times each philosopher has eaten. Also, experiment to see what happens if the `pick_up()` function is changed so that philosopher number 3 picks up the right chopstick first and everyone else picks up the left chopstick first.

9 How signals are handled is system-dependent. Try the following program on different systems to see what happens:

```
#include <stdio.h>
#include <signal.h>
#include <math.h>          /* HUGE_VAL defined here */

int main(void)
{
   double   x = HUGE_VAL, y = HUGE_VAL;

   signal(SIGFPE, SIG_IGN);
   printf("Ignore signal: x * y = %e\n", x * y);
   signal(SIGFPE, SIG_DFL);
   printf("Default signal: x * y = %e\n", x * y);
   return 0;
}
```

10 Starting with a variable of type `double **`, we showed how a matrix can be built dynamically and then passed to functions. Do we really have to do it all dynamically? Consider the code

```
int     i;
double     *a[3], det;
double     trace(double **, int);      /* function prototype */

for (i = 0; i < 3; ++i)
   a[i] = calloc(3, sizeof(double));   /* fill matrix a [][] */
.....
tr = trace(a, 3);
```

The array a is being passed as an argument, but from the function prototype we see that an argument of type `double **` is needed. Will your compiler complain? Explain.

11 In the 1970s, before C was available to us, we had occasion to use PL/1, a language that is commonly found on IBM mainframes. The PL/1 manuals always stressed that the compiler could create vectors and matrices that started at any desired index. For example, a declaration of the form

```
int   automobiles[1989 : 1999];    /* a PL/1 style declaration */
```

would create an array of size 11, with its index starting at 1989. This array could be used to store the number of automobiles sold, or projected to be sold, in the years 1989 to 1999. Perhaps one reason this concept was stressed was that FORTRAN and other languages were not able to create such arrays. In C, of course, such an array can be created. Explain how this is done.

12 Filling matrices with integers that are randomly distributed in the range from $-N$ to $+N$ is convenient for testing purposes. If you are checking machine computations by hand, then a reasonable value for N is 2 or 3. Write a function that fills matrices.

```
void fill_matrix(double **a, int m, int n, int N)
{
   .....
```

13 Let $A = (a_{ij})$ be an n × matrix. The determinant of A can be computed using Gaussian elimination. This algorithm requires that for $k = 1$ to n - 1 we do the following: Start with a_{kk} and examine all the elements in its column that are below a_{kk}, including a_{kk} itself. Among those elements find the one that is the largest in absolute value. This element is called the pivot. If the pivot is in the ith row, and i is not equal to k, then interchange the ith and kth rows. (Keep track of the number of interchanges that are performed.) After this has been done, a_{kk} will be the pivot. If this value is zero, then the value of the determinant is zero and the algorithm is finished. Otherwise, for i = k + 1 to n add a multiple of the kth row to the ith row, where the multiple is given by $(-1 / a_{kk}) \times a_{ik}$. The final step is to take the product of the diagonal elements of the matrix. If the number of row interchanges in the algorithm is even, then this product is the determinant. Otherwise, the negative of this product is the determinant. In this exercise you are to write a program that computes the determinant of n × n matrices. In order to have subscripts that start at 1 rather than 0, use the function `get_matrix_space()`. *Hint:* Use hand simulation of the algorithm on a small matrix first so that you understand the details of the algorithm.


```
for (k = 1; k <= n; ++k) {
   find the pivot element
   if the pivot is zero, return zero
   if the pivot is in the ith row, and i != k,
         then interchange the ith row with the kth row
         and increment the interchange counter i_cnt
   pivot = a[k][k];
   for (i = k + 1; i <= n; ++i) {
      multiplier = -a[i][k] * (1.0 / pivot);
      for (j = k; j <= n; ++j)
         a[i][j] += multiplier * a[k][j];
   }
}

if (i_cnt % 2 == 0)
   det = 1.0;
else
   det = -1.0;
for (i = 1; i <= n; ++i)
   det *= a[i][i];
return det;
```

14 One way to get a three-dimensional array is to make a declaration such as

```
double   a[9][2][7];
```

Suppose we want to pass this array to a function, and we want the function to work on three-dimensional arrays of various sizes. Because the function definition will require the 2 and the 7 to build the correct storage mapping function, implementing a as an array of arrays will not work. Instead, we can write

```
int      i, j, n1, n2, n3;
double   ***a;

.....                                  /* get sizes from somewhere */
a = calloc(n1, sizeof(int **));
for (i = 0; i < n1: ++i) {
   a[i] = calloc(n2, sizeof(double *));
   for (j = 0; j < n2; ++j)
      a[i][j] = calloc(n3, sizeof(double));
}
```

Draw a picture of how a can be thought of in memory. Write a program that implements this scheme. Your program should get the sizes from the user interactively, allocate space for the three-dimensional array, and fill the array with randomly distributed integers from a small range. Note that each `a[i]` can be thought of as a matrix. Print the array on the screen by first printing `a[0]`, then `a[1]`, and so forth.

Finally, print the sum of all the elements of the array. Use a function to compute this sum.

15 Consider the following program:

```
#include <stdio.h>

#define  N  3

double   trace(double *a[]);

int main(void)
{
  double   a[N][N] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

  printf("The trace of a is %.3f\n", trace(a));
  return 0;
}

double trace(double **a)
{
   int      i, j;
   double   sum = 0.0;;

   for (i = 0; i < N; ++i)     /* N is hardwired in the body */
      for (j = 0; j < N; ++j)
         sum += a[i][j];
   return sum;
}
```

The compiler complains about the call `trace(a)`. If we change this to

```
trace((double [N][N]) a)
```

will the compiler be happy? Will the code work? Explain.

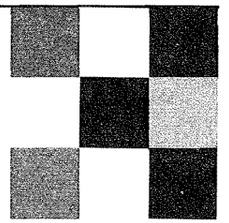
16 Remote procedure call (RPC) is used in distributed systems to execute code on other machines on a network. First, the program running on the local machine, called a *client*, sends a request to the second machine, called a *server*. The server calls a routine to perform the requested service. Finally, the results of the request are returned to the client. The following code is based on Sun Microsystems RPC and was written by Darrell Long:


```
/*
//SUN MICROSYSTEMS RPC code.
//   compile with cc pgc.c -lrpcsvc
*/

#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rstat.h>

/*
//Poll the host.
*/

void do_poll(char *host) {
   int     stat;
   struct  statstime result_stats;

   stat = callrpc(host, RSTATPROG, RSTATVERS_TIME,
                  RSTATPROC_STATS,
                  xdr_void, 0, xdr_statstime, &result_stats);
   if (stat == RPC_SUCCESS)
      fprintf(stdout, "DATA %s %ld %ld\n", host,
          result_stats.boottime.tv_sec,
          result_stats.curtime.tv_sec);
}

int main(void)
{
   do_poll("machine.school.edu");    /*p olled machine */
   return 0;
}
```

Run this code across your local network. Look in the file

/usr/include/rpcsvc/rstat.h

to figure out what these statistics mean.



Chapter 13

Moving from C to C++

This chapter gives an overview of the C++ programming language. It also provides an introduction to C++'s use as an object-oriented programming language. In the chapter, a series of programs is presented, and the elements of each program are carefully explained. The programs increase in complexity, and the examples in the later sections illustrate some of the concepts of object-oriented programming.

The examples in this chapter give simple, immediate, hands-on experience with key features of the C++ language. The chapter introduces the reader to stream I/O, operator and function overloading, reference parameters, classes, constructors, destructors, templates, and inheritance. Mastery of the individual topics requires a thorough reading of a companion book such as either Pohl, *C++ for C Programmers*, 2d ed (Redwood City, CA: Benjamin/Cummings, 1993) or Pohl, *Object-Oriented Programming Using C++*, 2d ed (Reading, MA: Addison-Wesley, 1997).

Object-oriented programming is implemented by the `class` construct. The `class` construct in C++ is an extension of `struct` in C. The later examples in this chapter illustrate how C++ implements OOP (object-oriented programming) concepts, such as data hiding, ADTs, inheritance, and type hierarchies.