The integer 3 in the declaration represents the number of elements in the array. The indexing of array elements always starts at 0.

A one-dimensional array declaration is a type followed by an identifier with a bracketed constant integral expression. The value of the expression, which must be positive, is the size of the array. It specifies the number of elements in the array. The array subscripts can range from 0 to *size* - 1. The lower bound of the array subscripts is 0 and the upper bound is *size* - 1. Thus, the following relationships hold:

```
int     a[size];  /* space for a[0], ... , a[size - 1] allocated */
```

lower bound = 0
upper bound = *size* - 1
size = *upper bound* + 1

It is good programming practice to define the size of an array as a symbolic constant.

```
#define   N   100

int   a[N];     /* space for a[0], ... , a[99] is allocated */
```

Given this declaration, the standard programming idiom for processing array elements is with a `for` loop. For example:

```
for (i = 0; i < N; ++i)
   sum += a[i];                /* process element a[i]
```

This iteration starts with the element `a[0]` and iteratively processes each element in turn. Because the termination condition is `i < N`, we avoid the error of falling off the end of the array. The last element processed is, correctly, `a[N - 1]`.

Initialization

Arrays may be of storage class automatic, external, or static, but not register. In traditional C, only external and static arrays can be initialized using an array initializer. In ANSI C, automatic arrays also can be initialized.

one_dimensional_array_initializer ::= { *initializer_list* }
initializer_list ::= *initializer* { , *initializer* }$_{0+}$
initializer ::= *constant_integral_expression*

Consider the example

```
float   f[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
```


This initializes `f[0]` to 0.0, `f[1]` to 1.0, and so on. When a list of initializers is shorter than the number of array elements to be initialized, the remaining elements are initialized to zero. For example,

```
int   a[100] = {0};
```

initializes all the elements of a to zero. If an external or static array is not initialized explicitly, then the system initializes all elements to zero by default. In contrast, automatic arrays are not necessarily initialized by the system. Although some compilers do this, others do not. Programmers should assume that uninitialized automatic arrays start with "garbage" values.

If an array is declared without a size and is initialized to a series of values, it is implicitly given the size of the number of initializers. Thus,

```
int a[] = {2, 3, 5, -7};        and        int a[4] = {2, 3, 5, -7};
```

are equivalent declarations. This feature works with character arrays as well. However, for character arrays an alternate notation is available. A declaration such as

```
char   s[] = "abc";
```

is taken by the compiler to be equivalent to

```
char   s[] = {'a', 'b', 'c', '\0'};
```

Subscripting

If a is an array, we can write `a[`*expr*`]`, where *expr* is an integral expression, to access an element of the array. We call *expr* a subscript, or index, of a. Let us assume that the declaration

```
int   i, a[N];
```

has been made, where `N` is a symbolic constant. The expression `a[i]` can be made to refer to any element of the array by assignment of an appropriate value to the subscript `i`. A single array element `a[i]` is accessed when `i` has a value greater than or equal to 0 and less than or equal to `N - 1`. If `i` has a value outside this range, a run-time error will occur when `a[i]` is accessed. Overrunning the bounds of an array is a common programming error. The effect of the error is system-dependent, and can be quite confusing. One frequent result is that the value of some unrelated variable will be returned or modified. It is the programmer's job to ensure that all subscripts to arrays stay within bounds.

Recall that a parenthesis pair () following an identifier tells the compiler that the identifier is a function name. Examples of this are `main()` and `f(a, b)`. In C, the parenthesis pair that follows an identifier is treated as an operator. In a similar fashion, the bracket pair [] also is treated as an operator. Both operators () and [] have the highest precedence and have left to right associativity.

6.2 Pointers

A simple variable in a program is stored in a certain number of bytes at a particular memory location, or address, in the machine. Pointers are used in programs to access memory and manipulate addresses.

If v is a variable, then &v is the location, or address, in memory of its stored value. The address operator & is unary and has the same precedence and right to left associativity as the other unary operators. Addresses are a set of values that can be manipulated. Pointer variables can be declared in programs and then used to take addresses as values. The declaration

```
int   *p;
```

declares p to be of type pointer to `int`. Its legal range of values always includes the special address 0 and a set of positive integers that are interpreted as machine addresses on the given C system. Some examples of assignment to the pointer p are

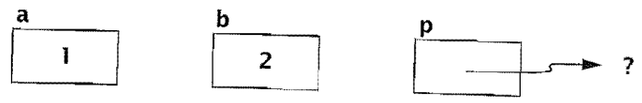
```
p = 0;
p = NULL;              /* equivalent to p = 0; */
p = &i;
p = (int *) 1776;      /* an absolute address in memory */
```

In the third example, we think of p as "referring to `i`" or "pointing to `i`" or "containing the address of `i`." In the fourth example, the cast is necessary to avoid a compiler error.

The indirection or dereferencing operator * is unary and has the same precedence and right to left associativity as the other unary operators. If p is a pointer, then *p is the value of the variable of which p is the address. The name "indirection" is taken from machine language programming. The direct value of p is a memory location, whereas *p is the indirect value of p—namely, the value at the memory location stored in p. In a certain sense * is the inverse operator to &. We want to give an explicit, yet elementary, example of how the pointer mechanism works. Let us start with the declaration


```
int   a = 1, b = 2, *p;
```

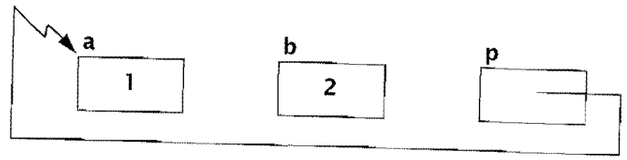
At this point, we can think of the variables a, b, and p stored in memory as



We think of the pointer p as an arrow, but because it has not yet been assigned a value, we do not know what it points to. Suppose that our next line of code is

```
p = &a;
```

We read this as "p is assigned the address of a," and we have the following picture:



Now let us make the assignment

```
b = *p;
```

We read this as "b is assigned the value pointed to by p." Because the pointer p points to a, the statement

```
b = *p;    is equivalent to    b = a;
```

Let us write a simple program that illustrates the distinction between a pointer value and its dereferenced value. We will use the %p format to print the value of a pointer, which on most systems produces a hexadecimal number. On ANSI C systems, the %p format is preferred. (See exercise 6, on page 312.)

In file locate.c

```
/* Printing an address, or location. */

#include <stdio.h>

int main(void)
{
   int   i = 7, *p = &i;

   printf("%s%d\n%s%p\n", "   Value of i: ", *p,
          "Location of i: ",  p);
   return 0;
}
```

The output of this program on our system is

```
   Value of i: 7
Location of i: effffb24
```

A pointer can be initialized in a declaration. The variable p is of type `int *` and its initial value is `&i`. Also, the declaration of `i` must occur before we take its address. The actual location of a variable in memory is system-dependent. The operator * dereferences p. That is, p contains an address, or location, and the expression *p has the value of what is stored at this location appropriately interpreted according to the type declaration of p.

The following table illustrates how some pointer expressions are evaluated:

Declarations and initializations		
`int i = 3, j = 5, *p = &i, *q = &j, *r;` `double x;`		
Expression	**Equivalent expression**	**Value**
`p == & i`	`p == (& i)`	1
`* * & p`	`* (* (& p))`	3
`r = & x`	`r = (& x)`	`/* illegal */`
`7 * * p / * q + 7`	`(((7 * (* p))) / (* q)) + 7`	11
`* (r = & j) *= * p`	`(* (r = (& j))) *= (* p)`	15

In this table, we attempted to assign `r` the value `&x`. Because `r` is a pointer to `int` and the expression `&x` is of type pointer to `double`, this is illegal. Also, note that in the table we used the expression


```
7 * * p / * q + 7
```

If instead we had written

```
7 * * p /* q + 7              /* trouble? */
```

we would find that the compiler treats /* as the start of a comment. This can result in a difficult bug.

In traditional C, conversions during assignment between different pointer types usually are allowed. In ANSI C, such conversions are not allowed unless one of the types is a pointer to `void`, or the right side is the constant 0. Thus, we can think of `void *` as a generic pointer type. This is an important point. (See Section 6.8, "Dynamic Memory Allocation With `calloc()` and `malloc()`," on page 262".)

Declarations	

`int *p;`
`float *q;`
`void *v;`	

Legal assignments	**Illegal assignments**
`p = 0;`	`p = 1;`
`p = (int *) 1;`	`v = 1;`
`p = v = q;`	`p = q;`
`p = (int *) q;`	

Of course, not every value is stored in an accessible memory location. It is useful to keep in mind the following prohibitions:

Constructs not to be pointed at

- Do not point at constants.
  ```
  &3                /* illegal */
  ```
- Do not point at ordinary expressions.
  ```
  &(k + 99)         /* illegal */
  ```
- Do not point at register variables.
  ```
  register   v;
  &v                /* illegal */
  ```

The address operator can be applied to variables and array elements. If a is an array, then expressions such as `&a[0]` and `&a[i+j+3]` make sense.

6.3 Call-by-Reference

Whenever variables are passed as arguments to a function, their values are copied to the corresponding function parameters, and the variables themselves are not changed in the calling environment. This "call-by-value" mechanism is strictly adhered to in C. To change the values of variables in the calling environment, other languages provide the "call-by-reference" mechanism. In this section we show how the use of addresses of variables as arguments to functions can produce the effect of "call-by-reference."

For a function to effect "call-by-reference," pointers must be used in the parameter list in the function definition. Then, when the function is called, addresses of variables must be passed as arguments. As an example of this, let us write a function that swaps the values of two variables in the calling environment.

```
void swap(int *p, int *q)
{
   int   tmp;

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

To test our function, we can write

```
#include <stdio.h>

void   swap(int *, int *);

int main(void)
{
   int    i = 3, j = 5;

   swap(&i, &j);
   printf("%d  %d\n", i, j);        /* 5  3 is printed */
   return 0;
}
```

Note that in the calling environment we have passed addresses as arguments to the function, and that in the function definition we have used pointers. This is a consistent pattern.

Dissection of the swap() Function

```
void swap(int *p, int *q)
{
   int   tmp;
```

This function takes two arguments of type pointer to `int` and returns nothing. The variable `tmp` is local to this function and is of type `int`. As the name indicates, we think of this as temporary storage.

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

The variable `tmp` is assigned the value pointed to by `p`. The object pointed to by `p` is assigned the value pointed to by `q`. The object pointed to by `q` is assigned the value `tmp`. This has the effect of interchanging in the calling environment the stored values of whatever `p` and `q` are pointing to.

The effect of "call-by-reference" is accomplished by

1 Declaring a function parameter to be a pointer

2 Using the dereferenced pointer in the function body

3 Passing an address as an argument when the function is called

6.4 The Relationship Between Arrays and Pointers

An array name by itself is an address, or pointer value, and pointers, as well as arrays, can be subscripted. Although pointers and arrays are almost synonymous in terms of how they are used to access memory, there are differences, and these differences are subtle and important. A pointer variable can take different addresses as values. In contrast, an array name is an address, or pointer, that is fixed.

Suppose that `a` is an array and that `i` is an `int`. It is a fundamental fact that the expression

```
a[i]        is equivalent to        *(a + i)
```

The expression `a[i]` has the value of the *i*th element of the array (counting from 0), whereas `*(a + i)` is the dereferencing of the expression `a + i`, a pointer expression that points `i` element positions past `a`. If `p` is a pointer, then in a similar fashion the expression

```
p[i]        is equivalent to        *(p + i)
```

This means that we can (and do) use array notation with pointers. Expressions such as `a + i` and `p + i` are examples of pointer arithmetic. The expression `a + i` has as its value the *i*th offset from the base address of the array `a`. That is, it points to the *i*th element of the array (counting from 0). In a similar manner, `p + i` is the *i*th offset from the value of `p`. The actual address produced by such an offset depends on the type that `p` points to.

When an array is declared, the compiler must allocate a sufficient amount of contiguous space in memory to contain all the elements of the array. The base address of the array is the initial location in memory where the array is stored; it is the address of the first element (index 0) of the array. Consider the following declarations:

```
#define   N   100

int   a[N], i, *p, sum = 0;
```

Suppose that the system assigns 300 as the base address of the array and that memory bytes numbered 300, 304, 308, ... , 696 are allocated as the addresses of `a[0]`, `a[1]`, `a[2]`, ... , `a[99]`, respectively. We are assuming that each byte is addressable and that 4 bytes are used to store an `int`. This is system-dependent. The statement

```
p = a;        is equivalent to        p = &a[0];
```

It causes 300 to be assigned to `p`. Pointer arithmetic provides an alternative to array indexing. The statement

```
p = a + 1;        is equivalent to        p = &a[1];
```

It causes 304 to be assigned to `p`. Assuming that the elements of `a` have been assigned values, we can use the following code to sum the array:

```
for (p = a; p < &a[N]; ++p)
   sum += *p;
```

First `p` is initialized to the base address of the array. In the body of the loop, the variable `sum` is incremented by the value pointed to by `p`. Each time through the loop, `p` is

incremented, so its successive values are `&a[0]`, `&a[1]`, ... , `&a[N-1]`. Here is another way of summing the array:

```
for (i = 0; i < N; ++i)
   sum += *(a + i);
```

In the body of the loop the pointer value `a` is offset by `i` and then dereferenced. This produces the value `a[i]`. Finally, here is a third way of summing the array:

```
p = a;
for (i = 0; i < N; ++i)
   sum += p[i];
```

Note that because `a` is a constant pointer, expressions such as

```
a = p          ++a          a += 2          &a
```

are illegal. We cannot change the value of `a`.

6.5 Pointer Arithmetic and Element Size

Pointer arithmetic is one of the powerful features of C. If the variable `p` is a pointer to a particular type, then the expression `p + 1` yields the correct machine address for storing or accessing the next variable of that type. In a similar fashion, pointer expressions such as `p + i` and `++p` and `p += i` all make sense. If `p` and `q` are both pointing to elements of an array, then `p - q` yields the `int` value representing the number of array elements between `p` and `q`. Even though pointer expressions and arithmetic expressions have a similar appearance, there is a critical difference in interpretation between the two types of expressions. The following code illustrates this:

```
double   a[2], *p, *q;

p = a;                              /* points to base of array */
q = p + 1;                          /* equivalent to q = &a[1] */
printf("%d\n", q - p);                      /* 1 is printed */
printf("%d\n", (int) q - (int) p);          /* 8 is printed */
```

On most machines, a `double` is stored in 8 bytes. Because `p` points to a `double` and `q` points to the next `double`, the difference in terms of array elements is 1, but the difference in memory locations is 8.

6.6 Arrays as Function Arguments

In a function definition, a formal parameter that is declared as an array is actually a pointer. Corresponding to this, when an array is passed as an argument to a function, the base address of the array is passed "call-by-value." The array elements themselves are not copied. As a notational convenience, the compiler allows array bracket notation to be used in declaring pointers as parameters. To illustrate these ideas, we write a function that sums the elements of an array of type `double`.

```
double sum(double a[], int n)      /* n is the size a[] */
{
   int      i;
   double   sum = 0.0;

   for (i = 0; i < n; ++i)
      sum += a[i];
   return sum;
}
```

Note carefully that in the header to the function definition the declaration of the parameter `a` as an array is equivalent to its declaration as a pointer. This means that an equivalent function definition is given by

```
double sum(double *a, int n)      /* n is the size a[] */
{
   .....
```

Although an array declaration is equivalent to a pointer declaration in a parameter list, this equivalence does *not* hold for external declarations or for declarations within the body of a function.

Suppose that in `main()` we have an array, or vector, `v` declared with 100 elements. After the elements of the vector have been assigned values, we can use the above function `sum()` to add various of the elements of `v`. The following table illustrates some of the possibilities:

Various ways that `sum()` might be called	
Invocation	What gets computed and returned
`sum(v, 100)`	`v[0] + v[1] + ...c... + v[99]`
`sum(v, 88)`	`v[0] + v[1] + ...c... + v[87]`
`sum(&v[7], k - 7)`	`v[7] + v[8] + ...c... + v[k - 1]`
`sum(v + 7, 2 * k)`	`v[7] + v[8] + ...c... + v[2 * k + 6]`

The last call illustrates again the use of pointer arithmetic. The base address of `v` is offset by 7, causing the local pointer variable `a` in `sum()` to be initialized to this value. This causes all address calculations inside the function call to be similarly offset.

6.7 An Example: Bubble Sort

Although we presented a bubble sort in Section 1.8, "Arrays, Strings, and Pointers," on page 37, we will do so again here and illustrate in some detail how the function works on a particular array of integers. We will use the function `swap()` from the previous section.

```
void   swap(int *, int *);

void bubble(int a[], int n)      /* n is the size of a[] */
{
   int    i, j;

   for (i = 0; i < n - 1; ++i)
      for (j = n - 1; j > i; --j)
         if (a[j-1] > a[j])
            swap(&a[j-1], &a[j]);
}
```

Because of pointer arithmetic, the expressions `a + i` and `&a[i]` are equivalent. Thus, the function call to `swap()` could have been written

```
swap(a + i, a + j);
```

Now we describe how the bubble sort works. Suppose that in `main()` we have

```
int   a[] = {7, 3, 66, 3, -5, 22, 77, 2};
bubble(a, 8);
```

The following table shows the elements of `a[]` after each pass of the outer loop in the function `bubble()`:

Unordered data:	7	3	66	3	-5	22	-77	2
First pass:	-77	7	3	66	3	-5	22	2
Second pass:	-77	-5	7	3	66	3	2	22
Third pass:	-77	-5	2	7	3	66	3	22
Fourth pass:	-77	-5	2	3	7	3	66	22
Fifth pass:	-77	-5	2	3	3	7	22	66
Sixth pass:	-77	-5	2	3	3	7	22	66
Seventh pass:	-77	-5	2	3	3	7	22	66

At the start of the first pass, `a[6]` is compared with `a[7]`. Because the values are in order, they are not exchanged. Then `a[5]` is compared with `a[6]`, and because these values are out of order, they are exchanged. Then `a[4]` is compared with `a[5]`, and so on. Adjacent out-of-order values are exchanged. The effect of the first pass is to "bubble" the smallest value in the array into the element `a[0]`. In the second pass `a[0]` is left unchanged and `a[6]` is compared first with `a[7]`, and so on. After the second pass, the next to the smallest value is in `a[1]`. Because each pass bubbles the next smallest element to its appropriate array position, the algorithm will, after $n - 1$ passes, have all the elements ordered. Notice that in this example all the elements have been ordered after the fifth pass. It is possible to modify the algorithm so that it terminates earlier by adding a variable that detects if no exchanges are made in a given pass. We leave this as an exercise.

A bubble sort is very inefficient. If the size of the array is n, then the number of comparisons performed is proportional to n^2. The merge sort discussed in Section 6.9, "An Example: Merge and Merge Sort," on page 266, is much more efficient; it is an $n \log n$ algorithm.

6.8 Dynamic Memory Allocation With `calloc()` and `malloc()`

C provides the functions `calloc()` and `malloc()` in the standard library, and their function prototypes are in *stdlib.h*. The name `calloc` stands for "contiguous allocation," and the name `malloc` stands for "memory allocation."

The programmer uses `calloc()` and `malloc()` to dynamically create space for arrays, structures, and unions. (See Chapter 9, "Structures and Unions," and Chapter 10, "Structures and List Processing," for structures and unions.) Here is a typical example that shows how `calloc()` can be used to allocate space dynamically for an array:

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

int main(void)
{
   int   *a;                          /* to be used as an array */
   int   n;                           /* the size of the array */
   .....                              /* get n from somewhere, perhaps
                                         interactively from the user */
   a = calloc(n, sizeof(int));        /* get space for a */
   .....                              /* use a as an array */
```

The function `calloc()` takes two arguments, both of type `size_t`. In ANSI C, `size_t` must be an unsigned integral type. Typically, a `typedef` is used in *stdlib.h* to make `size_t` equivalent to the type `unsigned int`. A function call of the form

```
calloc(n, el_size)
```

allocates contiguous space in memory for an array of `n` elements, with each element having *el_size* bytes. The space is initialized with all bits set to zero. If the call is successful, a pointer of type `void *` that points to the base of the array in memory is returned; otherwise, `NULL` is returned. Note that the use of the `sizeof` operator makes the code appropriate for machines having either 2- or 4-byte words.

The programmer uses `malloc()` in a similar fashion. This function takes a single argument of type `size_t`. If the call is successful, it returns a pointer of type `void *` that points to the requested space in memory; otherwise, `NULL` gets returned. Instead of writing

```
a = calloc(n, sizeof(int));
```

we could have written

```
a = malloc(n * sizeof(int));
```

Unlike `calloc()`, the function `malloc()` does not initialize the space in memory that it makes available. If there is no reason to initialize the array to zero, then the use of either `calloc()` or `malloc()` is acceptable. In a large program, `malloc()` may take less time.

Space that has been dynamically allocated with either `calloc()` or `malloc()` does not get returned to the system upon function exit. The programmer must use `free()` explicitly to return the space. A call of the form

```
free(ptr)
```

causes the space in memory pointed to by `ptr` to be deallocated. If `ptr` is `NULL`, the function has no effect. If `ptr` is not `NULL`, it must be the base address of space previously allocated by a call to `calloc()`, `malloc()`, or `realloc()` that has not yet been deallocated by a call to `free()` or `realloc()`. Otherwise, the call is in error. The effect of the error is system-dependent.

Let us write a small program that illustrates the ideas we have presented in this section. The first `printf()` statement in the program tells the user exactly what the program does.

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

void   fill_array(int *a, int n);
int    sum_array(int *a, int n);
void   wrt_array(int *a, int n);
```



```
int main(void)
{
   int   *a, n;

   srand(time(NULL));     /* seed the random number generator */
   printf("\n%s\n",
      "This program does the following repeatedly:\n"
      "\n"
      "   1  create space for an array of size n\n"
      "   2  fill the array with randomly distributed digits\n"
      "   3  print the array and the sum of its element\n"
      "   4  release the space\n");
   for ( ; ; ) {
      printf("Input n: ");
      if (scanf("%d", &n) != 1 || n < 1)
         break;
      putchar('\n');
      a = calloc(n, sizeof(int));   /* allocate space for a[] */
      fill_array(a, n);
      wrt_array(a, n);
      printf("sum = %d\n\n", sum_array(a, n));
      free(a);
   }
   printf("\nBye!\n\n");
   return 0;
}

void fill_array(int *a, int n)
{
   int   i;

   for (i = 0; i < n; ++i)
      a[i] = rand() % 19 - 9;
}

int sum_array(int *a, int n)
{
   int   i, sum = 0;

   for (i = 0; i < n; ++i)
      sum += a[i];
   return sum;
}
```

```
void wrt_array(int *a, int n)
{
   int   i;

   printf("a = [");
   for (i = 0; i < n; ++i)
      printf("%d%s", a[i], ((i < n - 1) ? ", " : "]\n"));
}
```

Offsetting the Pointer

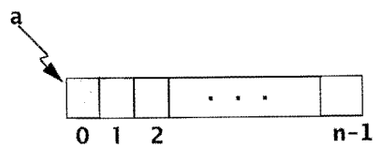
For arrays (vectors) intended for mathematical use, we often want to index the arrays from 1 instead 0. In a small program we can do something like the following:

```
int      n;
double   *a;

.....
a = calloc(n + 1, sizeof(double));
```

Because the size of a is one more than n, we can disregard a[0] and use a[1], ... , a[n] as needed. This is certainly an acceptable scheme. However, an alternative scheme is to do the following. First we write

```
a = calloc(n, sizeof(double));
```

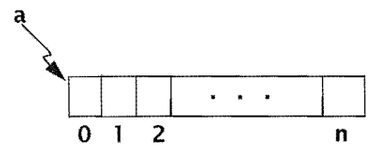
At this point, here is what we have in memory:



Next, we write

```
--a;          /* offset the pointer */
```


The picture of what is in memory changes to the following:



Notice that the shaded area indicates memory the programmer does not own. Thus, a[0] should not be accessed: neither written to nor read. The elements a[1], ... , a[n], however, are now accessible for use by the programmer. To deallocate the space, we can write free(a + 1).

In Section 12.6, "Dynamic Allocation of Matrices," on page 571, we will expand on the ideas presented in this section when we deal with mathematical matrices.

6.9 An Example: Merge and Merge Sort

Suppose we have two ordered arrays of integers, say a[] and b[]. If we want to merge them into another ordered array, say c[], then the algorithm to do so is simple. First, compare a[0] and b[0]. Whichever is smaller, say b[0], put into c[0]. Next, compare a[0] and b[1]. Whichever is smaller, say b[1], put into c[1]. Next, compare a[0] and b[2]. Whichever is smaller, say a[0], put into c[2]. Next, compare a[1] and b[2], and so on. Eventually one of the arrays a[] or b[] will be exhausted. At that point, the remainder of the elements in the other array are simply copied into c[]. Here is a function that does this, along with a header file that will get used in a program that tests our functions:

In file mergesort.h

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

void   merge(int a[], int b[], int c[], int m, int n);
void   mergesort(int key[], int n);
void   wrt(int key[], int sz);
```

In file merge.c

```
/* Merge a[] of size m and b[] of size n into c[]. */

#include "mergesort.h"

void merge(int a[], int b[], int c[], int m, int n)
{
   int   i = 0, j = 0, k = 0;

   while (i < m && j < n)
      if (a[i] < b[j])
         c[k++] = a[i++];
      else
         c[k++] = b[j++];
   while (i < m)                    /* pick up any remainder */
      c[k++] = a[i++];
   while (j < n)
      c[k++] = b[j++];
}
```

The array `c[]` is assumed to contain enough space to hold both `a[]` and `b[]`. The programmer must make certain that the bounds on `c[]` are not overrun. Observe that one, or possibly both, of the last two `while` statements that are used to pick up any remainder do not get executed. This is because at least one of the two conditions `i < m` and `j < n` will not be true.

Next, we will write a `mergesort()` function that calls `merge()`. In contrast to a bubble sort, a merge sort is very efficient. Our function `mergesort()` will act on an array `key[]`, which has a size that is a power of 2. The "power of 2" requirement will help to make the explanation simpler. In exercise 16, on page 315, we indicate how this restriction can be removed. To understand how merge sort works, let us suppose that `key[]` contains the following 16 integers:

```
4  3  1  67  55  8  0  4  -5  37  7  4  2  9  1  -1
```


The algorithm will work on the data in a number of passes. The following table shows how we want the data to look after each pass:

Unordered data:	4	3	1	67	55	8	0	4	-5	37	7	4	2	9	1	-1
First pass:	3	4	1	67	8	55	0	4	-5	37	4	7	2	9	-1	1
Second pass:	1	3	4	67	0	4	8	55	-5	4	7	37	-1	1	2	9
Third pass:	0	1	3	4	4	8	55	67	-5	-1	1	2	4	7	9	37
Fourth pass:	-5	-1	0	1	1	2	3	4	4	4	7	8	9	37	55	67

After the first pass, we want each successive pair of integers to be in order. After the second pass, we want each successive quartet of integers to be in order. After the third pass, we want each successive octet of integers to be in order. Finally, after the fourth pass, we want all 16 of the integers to be in order. At each stage `merge()` is used to accomplish the desired ordering. For example, after the third pass we have the two subarrays

```
0  1  3  4  4  8  55  67      and      -5  -1  1  2  4  7  9  37
```

which are both in order. By merging these two subarrays, we obtain the completely ordered array given in the last line of the table above. Surprisingly, the code that accomplishes this is quite short and it illustrates the power of pointer arithmetic.

In file mergesort.c

```
/* Mergesort: Use merge() to sort an array of size n. */

#include "mergesort.h"
void mergesort(int key[], int n)
{
   int   j, k, m, *w;

   for (m = 1; m < n; m *= 2)              /* m is a power of 2 */
      ;
   if (n < m) {
      printf("ERROR: Array size not a power of 2 - bye!\n");
      exit(1);
   }
   w = calloc(n, sizeof(int));             /* allocate workspace */
   assert(w != NULL);              /* check that calloc() worked */
   for (k = 1; k < n; k *= 2) {
      for (j = 0; j < n - k; j += 2 * k)
         /*
         // Merge two subarrays of key[] into a subarray of w[].
         */
         merge(key + j, key + j + k, w + j, k, k);
      for (j = 0; j < n; ++j)
         key[j] = w[j];                  /* write w back into key */
   }
   free(w);                                /* free the workspace */
}
```

Dissection of the mergesort() Function

```
for (m = 1; m < n; m *= 2)
   ;
```

After this loop, the value of m is the smallest power of 2 that is greater than or equal to n. If n is a power of 2, then n will be equal to m.

```
if (n < m) {
   printf("ERROR: Array size not a power of 2 - bye!\n");
   exit(1);
}
```

We test to see if n is a power of 2. If it is not a power of 2, we exit the program with an appropriate message.


```
w = cdalloc(n, sizeof(int));            /* allocate workspace */
```

The function calloc() is in the standard library, and its function prototype is in the header file *stdlib.h*. The function is used to allocate space in memory for an array dynamically. The first argument to the function specifies the number of elements in the array, and the second argument specifies the size in bytes needed to store each element. The function returns a pointer of type void * to the allocated space. Because its type is void *, the pointer can be assigned to w without a cast. In traditional C, we would have written

```
w = (int *) calloc(n, sizeof(int));
```

Although w is a pointer to int, we can now use it just as we would an array. Even though mergesort() is very efficient, the fact that additional workspace is needed to sort an array is a disadvantage. In Chapter 8 we will discuss the use of quicksort(), which does not require additional workspace.

```
assert(w != NULL);              /* check that calloc() worked */
```

If calloc() fails to allocate space—for example, the system free store (heap) is exhausted—then NULL is returned. We used the assert() macro in *assert.h* to make sure this did not happen. If the expression w != NULL is false, then the program will be aborted at this point.

```
for (k = 1; k < n; k *= 2) {
   for (j = 0; j < n - k; j += 2 * k)
      /*
      // Merge two subarrays of key[] into a subarray of w[].
      */
      merge(key + j, key + j + k, w + j, k, k);
   for (j = 0; j < n; ++j)
      key[j] = w[j];                  /* write w back into key */
}
```

This is the heart of the algorithm. Suppose we start with key[] having the data given as unordered in the table above. In the first pass of the outer loop, k is 1. Consider the first inner loop

```
for (j = 0; j < n - k; j += 2 * k)
   /*
   // Merge two subarrays of key[] into a subarray of w[].
   */
   merge(key + j, key + j + k, w + j, k, k);
```

The first call to `merge()` is equivalent to

```
merge(key + 0, key + 0 + 1, w + 0, 1, 1);
```

The arrays based at `key` and `key + 1`, both of size 1, are being merged and put into a subarray of size 2 of `w` that is based at `w + 0`. This will result in `w[0]` and `w[1]` being in order. The next call to `merge()` is equivalent to

```
merge(key + 2, key + 2 + 1, w + 2, 1, 1);
```

The arrays based at `key + 2` and `key + 3`, both of size 1, are being merged and put into a subarray of size 2 of `w` that is based at `w + 2`. This will result in `w[2]` and `w[3]` being in order. The next call ... , and so on. After the first pass of the outer loop, each successive pair of elements in `w[]` is in order; see the preceding table. At this point, the array `w[]` is copied into `key[]`.

In the second pass of the outer `for` loop, `k` is 2, and the next call to `merge()` is equivalent to the following:

```
merge(key + 0, key + 0 + 2, w + 0, 2, 2);
```

The arrays based at `key` and `key + 2`, both of size 2, are being merged and put into a subarray of size 4 of `w` that is based at `w + 0`. This will result in `w[0]`, `w[1]`, `w[2]`, `w[3]` being in order. The next call to `merge()` is equivalent to

```
merge(key + 4, key + 4 + 2, w + 4, 2, 2);
```

The arrays based at `key + 4` and `key + 6`, both of size 2, are being merged and put into a subarray of size 4 of `w` based at `w + 4`. This will result in `w[4]`, `w[5]`, `w[6]`, `w[7]` being in order. The next call ... , and so on. After the second pass of the outer loop, each successive quartet of elements in `w[]` is in order; see the preceding table. At this point the array `w[]` is copied into `key[]`. In the third pass of the outer loop, `k` is 4, and the next call to `merge()` ... , and so on.

■ `free(w);` `/* free the workspace */`

This function is in the standard library, and its function prototype is in *stdlib.h*. It causes the space pointed to by `w` to be deallocated. The system is then able to use this space for some other purpose. Unlike automatic storage, space allocated by `calloc()` or `malloc()` is not relinquished automatically upon exit from a function. The programmer must explicitly free this space.

We want to write a program to test the functions `merge()` and `mergesort()`. The interested reader should modify the program to print out the array `key[]` after each pass of the outer loop in `mergesort()` to check that the preceding table is reproduced. Here are the remaining two functions that comprise our program:

In file main.c

```
/* Test merge() and mergesort(). */

#include "mergesort.h"

int main(void)
{
   int   sz, key[] = { 4, 3, 1, 67, 55, 8, 0, 4,
                       -5, 37, 7, 4, 2, 9, 1, -1 };

   sz = sizeof(key) / sizeof(int);   /* the size of key[] */
   printf("Before mergesort:\n");
   wrt(key, sz);
   mergesort(key, sz);
   printf("After mergesort:\n");
   wrt(key, sz);
   return 0;
}
```

In file wrt.c

```
#include "mergesort.h"

void wrt(int key[], int sz)
{
   int   i;

   for (i = 0; i < sz; ++i)
      printf("%4d%s", key[i], ((i < sz - 1) ? "" : "\n"));
}
```

The amount of work that a merge sort does when sorting n elements is proportional to $n \log n$. Compared to a bubble sort, this is a very significant improvement. Sorting is often critical to the efficient handling of large amounts of stored information. However, it is beyond the scope of this text to discuss the topic in detail; see *The Art of Computer Programming*, vol. 3, *Sorting and Searching*, by Donald Ervin Knuth (Reading, MA: Addison-Wesley, 1973).

6.10 Strings

Strings are one-dimensional arrays of type `char`. By convention, a string in C is terminated by the end-of-string sentinel `\0`, or null character. The null character is a byte with all bits off; hence, its decimal value is zero. Because dealing with strings has its own flavor, we treat the topic separately. It is useful to think of strings as having a variable length, delimited by `\0`, but with a maximum length determined by the size of the string. The size of a string *must* include the storage needed for the end-of-string sentinel. As with all arrays, it is the job of the programmer to make sure that string bounds are not overrun.

String constants are written between double quotes. For example, `"abc"` is a character array of size 4, with the last element being the null character `\0`. Note that string constants are different from character constants. For example, `"a"` and `'a'` are not the same. The array `"a"` has two elements, the first with value `'a'` and the second with value `'\0'`.

A string constant, like an array name by itself, is treated by the compiler as a pointer. Its value is the base address of the string. Consider the following code:

```
char   *p = "abc";

printf("%s %s\n", p, p + 1);      /* abc bc is printed */
```

The variable `p` is assigned the base address of the character array `"abc"`. When a pointer to `char` is printed in the format of a string, the pointed-at character and each successive character are printed until the end-of-string sentinel is reached. Thus, in the `printf()` statement, the expression `p` causes `abc` to be printed, and the expression `p + 1`, which points to the letter `b` in the string `"abc"`, causes `bc` to be printed. Because a string constant such as `"abc"` is treated as a pointer, expressions such as

```
"abc"[1]        and        *("abc" + 2)
```

are possible. (See exercise 22, on page 317.) Such expressions are not used in serious code, but they help to emphasize that string constants are treated as pointers.

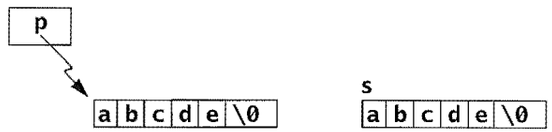
As we have already seen, arrays and pointers have similar uses. They also have differences. Let us consider the two declarations

```
char  *p = "abcde";        and        char  s[] = "abcde";
```


In the first declaration, the compiler allocates space in memory for `p`, puts the string constant `"abcde"` in memory somewhere else, and initializes `p` with the base address of the string constant. We now think of `p` as pointing to the string. The second declaration is equivalent to

```
char   s[] = {'a', 'b', 'c', 'd', 'e', '\0'};
```

Because the brackets are empty, the compiler allocates 6 bytes of memory for the array `s`. The first byte is initialized with `'a'`, the second byte is initialized with `'b'`, and so on. Here is how we think of these objects stored in memory:



A `char` is always stored in 1 byte, and on most machines a pointer is stored in a word. Thus, on our machine, `p` is stored in 4 bytes, and `s` is stored in 6 bytes. Of course, `p` contains the address of a string that requires another 6 bytes of storage. (See exercise 24, on page 319, for further discussion.)

For technical reasons, it is better not to print null characters. If null characters are put into a file and then another utility is used to process the file, the result can be confusing. (See exercise 25, on page 319.) However, the printing of null strings is perfectly acceptable. One natural instance of this occurs when dealing with plurals:

```
char   *s;
int    nfrogs;
.....                                  /* get nfrogs from somewhere */
s = (nfrogs == 1) ? "" : "s";
printf("We found %d frog%s in the pond.\n", nfrogs, s);
```

To illustrate string processing, we will write a function that counts the number of words in a string. We assume that words in the string are separated by white space. Our function will use the macro `isspace()`, which is defined in the standard header file *ctype.h*. This macro is used to test whether a character is a blank, tab, newline, or some other white-space character. If the argument is a white-space character, then a nonzero (*true*) value is returned; otherwise, zero (*false*) is returned.

```
/* Count the number of words in a string. */

#include <ctype.h>

int word_cnt(const char *s)
{
   int   cnt = 0;

   while (*s != '\0') {
      while (isspace(*s))                    /* skip white space */
         ++s;
      if (*s != '\0') {                       /* found a word */
         ++cnt;
         while (!isspace(*s) && *s != '\0')   /* skip the word */
            ++s;
      }
   }
   return cnt;
}
```

This is a typical string-processing function. Pointer arithmetic and dereferencing are used to search for various characters or patterns. Often a character pointer is used to march along a string while parsing it or interpreting it in some way.

6.11 String-Handling Functions in the Standard Library

ANSI C provides the programmer with numerous string-handling functions; see Appendix A, "The Standard Library," for details. In this section we want to illustrate the use of a few selected string-handling functions and describe how they work. The function prototypes for string-handling functions are given in the standard header file *string.h*.

We will see that some of the parameters in the function definitions of the string-handling functions have the type `const char *`. The type qualifier `const` in this context is telling the compiler that the character pointed to in memory should not be changed. (See exercise 28, on page 321.) The pointer itself, however, can be changed.

Some String-Handling Functions in the Standard Library

- `char *strcat(char *s1, const char *s2);`

This function takes two strings as arguments, concatenates them, and puts the result in `s1`. The programmer must ensure that `s1` points to enough space to hold the result. The string `s1` is returned.

- `int strcmp(const char *s1, const char *s2);`

Two strings are passed as arguments. An integer is returned that is less than, equal to, or greater than zero, depending on whether `s1` is lexicographically less than, equal to, or greater than `s2`.

- `char *strcpy(char *s1, const char *s2);`

The characters in the string `s2` are copied into `s1` until `\0` is moved. Whatever exists in `s1` is overwritten. It is assumed that `s1` has enough space to hold the result. The pointer `s1` is returned.

- `size_t strlen(const char *s);`

A count of the number of characters before `\0` is returned. ANSI C requires the type `size_t` to be an integral unsigned type. Typically, on systems with 4-byte `int`s it is equivalent to `unsigned int`, and on systems with 2-byte `int`s it is equivalent to `unsigned long`.

There is nothing special about these functions. They can be written in C, and are all quite short. Variables in them are often declared to have storage class `register` in an attempt to make them execute more quickly. Here is one way the function `strlen()` could be written:

```
size_t strlen(const char *s)
{
   size_t   n;

   for (n = 0; *s != '\0'; ++s)
      ++n;
   return n;
}
```

The loop continues counting until the end-of-string character '\0' is detected. For example, strlen("abc") returns the value 3, and strlen("") returns the value 0.

The function definition for strcpy() is also simple, but it contains a C idiom that needs to be explained in detail.

```
char *strcpy(char *s1, register const char *s2)
{
   register char   *p = s1;

   while (*p++ = *s2++)
      ;
   return s1;
}
```

Dissection of the strcpy() Function

■ `register char   *p = s1;`

Observe that p is being initialized, not *p. The pointer p is initialized to the pointer value s1. Thus, p and s1 point to the same memory location.

■ `while (*p++ = *s2++)`
`   ;`

As long as the expression *p++ = *s2++ is *true* (not zero), the body of the while loop gets executed. The body, however, is empty, so its execution has no effect. All the action takes place in the side effects of the expression controlling the while loop.

■ `*p++`

Because the unary operator ++ is in postfix position, it has higher precedence than the * operator. Thus, *p++ is equivalent to *(p++), which means that p itself is being incremented. In contrast, the expression (*p)++ would increment what p is pointing to, leaving the value of p itself unchanged. (See exercise 17, on page 316.) The value of the expression p++ is the current value of p. This value is dereferenced; then the value of p in memory is incremented, causing it to point to the next character in the string.

■ `*p++ = *s2++`

Suppose p is pointing to a character in one string, and suppose s2 is pointing to a character in a different string. (See exercise 18, on page 316, for another scenario.) The value pointed to by s2 is assigned to the value pointed to by p, and this is the value of the expression as a whole. Then both p and s2 get incremented, which causes them to point to the next character in each of the two strings.

■ `while (*p++ = *s2++)`
`   ;`

The character pointed to by p gets assigned the value of the character pointed to by s2. Then both p and s2 get incremented so that they each point to the next character. This process continues until s2 points to the null character \0, which has the value 0. At that point the value 0 is assigned to what p is pointing to, putting an end-of-string sentinel in place, and the value of the expression as a whole is 0, causing the flow of control to exit the while loop. The effect of all of this is to copy the string s2 into the string s1.

The function strcat() is also quite simple. It can be written in a number of ways. Here is one of them:

```
char *strcat(char *s1, register const char *s2)
{
   register char   *p = s1;

   while (*p)                /* go to the end */
      ++p;
   while (*p++ = *s2++)              /* copy */
      ;
   return s1;
}
```

Dissection of the `strcat()` Function

- `register char *p = s1;`

Observe that `p` is being initialized, not `*p`. The pointer `p` is initialized to the pointer value `s1`. Thus, `p` and `s1` point to the same memory location.

-
```
while (*p)
   ++p;
```

As long as the value pointed to by `p` is nonzero, `p` is incremented, causing it to point at the next character in the string. When `p` points to the end-of-string sentinel `\0`, the expression `*p` has the value 0. This causes control to exit the `while` statement. Notice that we could have written

`while (*p != '\0')` instead of `while (*p)`

to achieve the same effect. Functions often are written in a sparse manner. Although one might think that sparse C code produces faster running object code, most C compilers produce the same object code for both forms of this `while` statement.

-
```
while (*p++ = *s2++)
   ;
```

At the beginning of this `while` statement, `p` points to the null character at the end of the string pointed to by `s1`. Thus, the characters in `s2` get copied one after another into memory, overwriting the null character at the end of `s1` and whatever follows it. The overall effect of this `while` statement is to concatenate `s2` to the end of `s1`.

String-handling functions are illustrated in the next table. Note carefully that it is the programmer's responsibility to allocate sufficient space for the strings that are passed as arguments to these functions.

Declarations and initializations	
`char s1[] = "beautiful big sky country", s2[] = "how now brown cow";`	
Expression	**Value**
`strlen(s1)`	25
`strlen(s2 + 8)`	9
`strcmp(s1, s2)`	*negative integer*
Statements	**What gets printed**
`printf("%s", s1 + 10);`	`big sky country`
`strcpy(s1 + 10, s2 + 8);`	
`strcat(s1, "s!");`	
`printf("%s", s1);`	`beautiful brown cows!`

6.12 Multidimensional Arrays

The C language allows arrays of any type, including arrays of arrays. With two bracket pairs, we obtain a two-dimensional array. This idea can be iterated to obtain arrays of higher dimension. With each bracket pair, we add another array dimension.

Examples of declarations of arrays	Remarks
`int a[100];`	a one-dimensional array
`int b[2][7];`	a two-dimensional array
`int c[5][3][2];`	a three-dimensional array

A k-dimensional array has a size for each of its k dimensions. If we let s_i represent the size of its ith dimension, then the declaration of the array will allocate space for $s_1 \times s_2 \times \ldots \times s_k$ elements. In the preceding table, `b` has 2×7 elements, and `c` has $5 \times 3 \times 2$ elements. Starting at the base address of the array, all the array elements are stored contiguously in memory. *Caution:* The multidimensional array mechanism

described here is often unsuitable for dealing with mathematical matrices. (See Section 12.6, "Dynamic Allocation of Matrices," on page 571, for further discussion.)

Two-dimensional Arrays

Even though array elements are stored contiguously one after the other, it is usually convenient to think of a two-dimensional array as a rectangular collection of elements with rows and columns. For example, if we declare

```
int   a[3][5];
```

then we can think of the array elements arranged as follows:

	col 1	col 2	col 3	col 4	col 5
row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

Because of the relationship between arrays and pointers, there are numerous ways to access elements of a two-dimensional array.

Expressions equivalent to a[i][j]
*(a[i] + j)
(*(a + i))[j]
(((a + i)) + j)
*(&a[0][0] + 5*i + j)

The parentheses are necessary because the brackets [] have higher precedence than the indirection operator *. We can think of a[i] as the *i*th row of a (counting from 0), and we can think of a[i][j] as the element in the *i*th row, *j*th column of the array (counting from 0). The array name a by itself is equivalent to &a[0]; it is a pointer to an array of 5 ints. The base address of the array is &a[0][0], not a. Starting at the base address of the array, the compiler allocates contiguous space for 15 ints.

The Storage Mapping Function

For any array, the mapping between pointer values and array indices is called the *storage mapping function*. Consider the declaration

```
int   a[3][5];
```

For the array a, the storage mapping function is specified by noting that

a[i][j] is equivalent to *(&a[0][0] + 5*i + j)

Formal Parameter Declarations

When a multidimensional array is a formal parameter in a function definition, all sizes except the first must be specified so that the compiler can determine the correct storage mapping function. (Because of this, multidimensional arrays are often inappropriate for use in mathematical programming; see Section 12.6, "Dynamic Allocation of Matrices," on page 571.) Consider the declaration

```
int   a[3][5];
```

This allocates space for a 3×5 matrix. Its row size is 3, and its column size is 5. After the elements of the matrix a have been assigned values, the following function can be used to sum the elements of the matrix. Note carefully that the column size *must* be specified.

```
int sum(int a[][5])
{
   int   i, j, sum = 0;

   for (i = 0; i < 3; ++i)
      for (j = 0; j < 5; ++j)
         sum += a[i][j];
   return sum;
}
```

In the header of the function definition, the following three parameter declarations are equivalent:

```
int a[][5]          int a[3][5]          int (*a)[5]
```

In the second declaration, the constant 3 acts as a reminder to human readers of the code, but the compiler disregards it. In the third declaration, the parentheses are neces-

sary because of operator precedence. *Caution:* These three declarations are equivalent only in a header to a function definition.

These ideas are related to what happens with respect to one-dimensional arrays. Recall that in a header to a function definition, the following three parameter declarations are equivalent:

```
int b[]          int b[3]          int *b
```

In the second declaration, the 3 serves as a reminded to human readers of the code, but the compiler disregards it. *Caution:* These three declarations are equivalent only in a header to a function definition.

There is nothing special about the type `int`. For example, in the header to a function definition the following three declarations are equivalent:

```
char *argv[]        char *argv[3]        char **argv
```

Although one bracket pair can be replaced by a *, the rule does not generalize. For example, in a header to a function definition

```
char x[][]      is not equivalent to      char **x
```

Three-dimensional Arrays

Arrays of dimension higher than two work in a similar fashion. Let us describe how three-dimensional arrays work. If we declare

```
int   a[7][9][2];
```

then the compiler will allocate space for $7 \times 9 \times 2$ contiguous `int`s. The base address of the array is `&a[0][0][0]`, and the storage mapping function is specified by noting that

```
a[i][j][k]      is equivalent to      *(&a[0][0][0] + 9*2*i + 2*j + k)
```

If an expression such as `a[i][j][k]` is used in a program, the compiler uses the storage mapping function to generate object code to access the correct array element in memory. Here is a function that will sum the elements of the array. Note carefully that all the sizes except the first *must* be specified.


```
int sum(int a[][9][2])
{
   int   i, j, k, sum = 0;

   for (i = 0; i < 7; ++i)
      for (j = 0; j < 9; ++j)
         for (k = 0; k < 2; ++k)
            sum += a[i][j][k];
   return sum;
}
```

In the header of the function definition, the following three parameter declarations are equivalent:

```
int a[][9][2]        int a[7][9][2]        int (*a)[9][2]
```

In the second declaration, the constant 7 acts as a reminder to human readers of the code, but the compiler disregards it. The other two constants are needed by the compiler to generate the correct storage mapping function. *Caution:* These three declarations are equivalent only in a header to a function definition.

Initialization

There are a number of ways to initialize a multidimensional array. Let us begin our discussion by considering the following three initializations, which are equivalent:

```
int   a[2][3] = {1, 2, 3, 4, 5, 6};
int   a[2][3] = {{1, 2, 3}, {4, 5, 6}};
int   a[ ][3] = {{1, 2, 3}, {4, 5, 6}};
```

If there are no inner braces, then each of the array elements `a[0][0]`, `a[0][1]`, ... , `a[1][2]` is initialized in turn. Note that the indexing is by rows. If there are fewer initializers than elements in the array, then the remaining elements are initialized to zero. If the first bracket pair is empty, then the compiler takes the size from the number of inner brace pairs. In any case, all sizes except the first must always be given explicitly.

If we are willing to put in all the braces, then we do not have to specify all the zeros. Consider the initialization

```
int   a[2][2][3] = {
         {{1, 1, 0}, {2, 0, 0}},
         {{3, 0, 0}, {4, 4, 0}}
      };
```

An equivalent initialization is given by

```
int   a[][2][3] = {{{1, 1}, {2}}, {{3}, {4, 4}}};
```

If the initializers are fully and consistently braced, then wherever there are not enough initializers listed, the remaining elements are initialized to zero.

In general, if an array of storage class automatic is not explicitly initialized, then array elements start with "garbage" values. Here is a simple way to initialize all array elements to zero:

```
int   a[2][2][3] = {0};        /* all element initialized to zero */
```

Unlike automatic arrays, all static and external arrays are initialized to zero by default.

The Use of typedef

Let us illustrate the use of `typedef` by defining a small number of functions that operate on vectors and matrices.

```
#define   N   3      /* the size of all vectors and matrices */

typedef   double   scalar;
typedef   scalar   vector[N];
typedef   scalar   matrix[N][N];
```

We have used the `typedef` mechanism to create the types `scalar`, `vector`, and `matrix`, which is both self-documenting and conceptually appropriate. Our programming language has been extended in a natural way to incorporate these new types as a domain. Notice how `typedef` can be used to build hierarchies of types. For example, we could have written

```
typedef   vector   matrix[N];
```

in place of

```
typedef   scalar   matrix[N][N];
```

The use of `typedef` to create type names such as `scalar`, `vector`, and `matrix` allows the programmer to think in terms of the application. Now we are ready to create functions that provide operations over our domain.


```
void add(vector x, vector y, vector z)          /* x = y + z */
{
   int i;

   for (i = 0; i < N; ++i)
      x[i] = y[i] + z[i];
}

scalar dot_product(vector x, vector y)
{
   int      i;
   scalar   sum = 0.0;

   for (i = 0; i < N; ++i)
      sum += x[i] * y[i];
   return sum;
}

void multiply(matrix a, matrix b, matrix c)     /* a = b * c */
{
   int   i, j, k;

   for (i = 0; i < N; ++i) {
      for (j = 0; j < N; ++j) {
         a[i][j] = 0.0;
         for (k = 0; k < N; ++k)
            a[i][j] += b[i][k] * c[k][j];
      }
   }
}
```

These routines are not generic because they work only with vectors and matrices of a fixed size. We show how this limitation can be removed in Section 12.6, "Dynamic Allocation of Matrices," on page 571.

6.13 Arrays of Pointers

In C, array elements can be of any type, including a pointer type. Arrays of pointers have many uses. In this section we will write a program that uses an array of pointers to sort words in a file lexicographically. As we will see, an array with elements of type `char *` can be thought of as an array of strings.

To explain how our program works, we need to imagine first how we are going to test it. Let us create the file *input* with the following two lines in it:

```
A is for apple or alphabet pie
which all get a slice of, come taste it and try.
```

To sort the words in this file, we will give the command

```
sort_words  <  input
```

This is supposed to cause the following to be printed on the screen:

```
A
a
all
alphabet
.....
which
```

Now we are ready to write our program. It will consist of header file and a number of *.c* files. Here is the header file and the function `main()`:

In file sort.h

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

#define   MAXWORD   50          /* max word size */
#define   N         300         /* array size of w[] */
```



```
void   error_exit_calloc_failed(void);
void   error_exit_too_many_words(void);
void   error_exit_word_too_long(void);
void   sort_words(char *w[], int n);
void   swap(char **p, char **q);
void   wrt_words(char *w[], int n);
```

In file main.c

```
/* Sort words lexicographically. */

#include "sort.h"

int main(void)
{
   char   word[MAXWORD];      /* work space */
   char   *w[N];              /* an array of pointers */
   int    n;                  /* number of words to be sorted */
   int    i;

   for (i = 0; scanf("%s", word) == 1; ++i) {
      if (i >= N)
         error_exit_too_many_words();
      if (strlen(word) >= MAXWORD)
         error_exit_word_too_long();
      w[i] = calloc(strlen(word) + 1, sizeof(char));
      if (w[i] == NULL)
         error_exit_calloc_failed();
      strcpy(w[i], word);
   }
   n = i;
   sort_words(w, n);          /* sort the words */
   wrt_words(w, n);           /* write sorted list of words */
   return 0;
}
```

Dissection of main( ) in the *sort_words* Program

```
#define   MAXWORD   50          /* max word size */
#define   N         300         /* array size of w[] */
```

These two lines in the header file define the symbolic constants `MAXWORD` and `N`, which get used in `main()` and in other functions. We are assuming that no word in the input file has more than 50 characters and that there are at most 300 words in the file.

```
char    word[MAXWORD];      /* work space */
char    *w[N];              /* an array of pointers */
```

We will use the array word to temporarily store each word that is read from the input file, and we expect each word to have less than MAXWORD characters. The declaration

```
char    *w[N];      is equivalent to      char    *(w[N]);
```

Thus, w is an array of pointers to char. This array will eventually store all the words that we read from the input file.

```
for (i = 0; scanf("%s", word) == 1; ++i) {
    .....
```

As long as scanf() is able to read characters from the standard input file and store them in word, the body of the for loop is executed. Recall that scanf() takes as arguments a control string followed by *addresses* that are matched with the formats in the control string. Here, the second argument to scanf() is the array name word by itself. Because an array name by itself is treated an address, we do not need to write &word. Technically, it would be an error to do so. Some compilers will warn the programmer about this kind of error; others will not.

```
if (i >= N)
    error_exit_too_many_words();
if (strlen(word) >= MAXWORD)
    error_exit_word_too_long();
```

If there are too many words, we print an error message and exit. If there are too many characters in the word read in by scanf(), we print an error message and exit. By calling error functions to do this, we reduce the clutter in main().

```
w[i] = calloc(strlen(word) + 1, sizeof(char));
```

The function calloc() is in the standard library, and its function prototype is in *stdlib.h*. A function call such as

```
calloc(n, sizeof(...))
```

dynamically allocates space for an array of n elements, with each element requiring sizeof(...) bytes in memory, and returns a pointer to the allocated space. The minimum number of bytes needed to store the string word, including the end-of-string sentinel \0, is strlen(word) + 1. (The + 1 is important, and unfortunately it is easy to forget.) The function calloc() dynamically allocates this space and returns a pointer to it. The pointer is assigned to w[i].

We could have invoked malloc() instead of calloc(). Either of the following statements would have worked just as well:

```
w[i] = malloc((strlen(word) + 1) * sizeof(char));
w[i] = malloc((strlen(word) + 1));
```

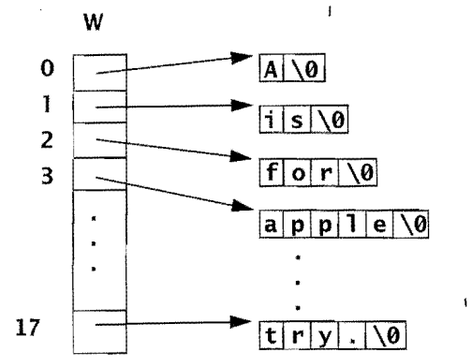
The statements are equivalent because sizeof(char) has value 1. The reason for writing it is to remind readers of the code that space for an array of chars is being requested. With calloc(), the allocated space is initialized to zero, but with malloc() no initialization takes place. Because we are going to overwrite the space, initialization is not needed.

```
if (w[i] == NULL)
    error_exit_calloc_failed();
```

Good programming practice requires that we check the return value from calloc() and malloc(). If the requested memory is unavailable, NULL is returned. If that is the case, we print an error message and exit.

```
strcpy(w[i], word);
```

After space has been allocated, the function strcpy() is used to copy word into memory starting at the address w[i]. We can think of w as an array of words.




```
sort_words(w, n);           /* sort the words */
wrt_words(w, n);            /* write sorted list of words */
```

The function sort_words() is used to sort the words in the array w lexicographically. Then wrt_words() is used to write the sorted list of words on the screen.

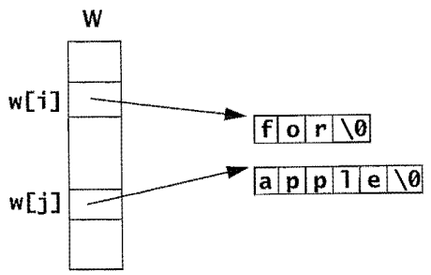
Now we want to look at the function `sort_words()`. It uses a transposition sort that is similar in flavor to a bubble sort.

In file sort.c

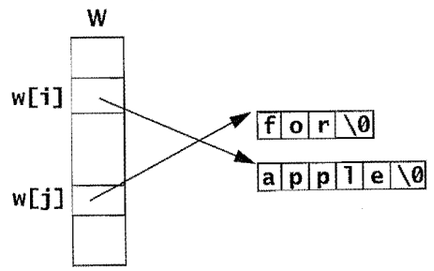
```
#include "sort.h"

void sort_words(char *w[], int n) /* n elements are to be sorted */
{
   int   i, j;

   for (i = 0; i < n; ++i)
      for (j = i + 1; j < n; ++j)
         if (strcmp(w[i], w[j]) > 0)
            swap(&w[i], &w[j]);
}
```

Notice that `strcmp()` is used to compare the two strings pointed to by `w[i]` and `w[j]`. If they are out of order, we use the function `swap()` to interchange the two pointer values. The addresses of the pointers are passed so that the pointer values themselves can be changed in the calling environment by the function call. The underlying character strings in memory do not get swapped; only the pointers to them are interchanged.

Before swapping



After swapping



In file swap.c

```
#include "sort.h"

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

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

Each of the two arguments `&w[i]` and `&w[j]` that get passed to `swap()` is an address of a pointer to `char`, or equivalently, a pointer to pointer to `char`. Hence, the formal parameters in the function definition of `swap()` are of type `char **`. The `swap()` function is very similar to the one we wrote previously; only the types are different. Consider the statement

```
tmp = *p;
```

Because `p` is a pointer to pointer to `char`, the expression `*p` that dereferences `p` is of type `char *`. Hence, we declared `tmp` to be of type `char *`.

In file error.c

```
#include "sort.h"

void error_exit_calloc_failed(void)
{
   printf("%s",
      "ERROR: The call to calloc() failed to\n"
      "       allocate the requested memory - bye!\n");
   exit(1);
}

void error_exit_too_many_words(void)
{
   printf("ERROR: At most %d words can be sorted - bye!\n", N);
   exit(1);
}
```

```
void error_exit_word_too_long(void)
{
   printf("%s%d%s",
      "ERROR: A word with more than ", MAXWORD, "\n"
      "       characters was found - bye!\n");
   exit(1);
}
```

In file wrt.c

```
#include "sort.h"

void wrt_words(char *w[], int n)
{
   int   i;

   for (i = 0; i < n; ++i)
      printf("%s\n", w[i]);
}
```

Of course, if the *sort_words* program were intended for serious work on a large amount of data, we would use a more efficient sorting algorithm. We could, for example, modify our `merge()` and `merge_sort()` functions to work on arrays of pointers to `char` rather than on arrays of `int`s. (See exercise 37, on page 325.) Also, we would want to count the words in the file and then allocate space for `w[]` dynamically, rather than using some arbitrarily large value `N` for the array size. (See Chapter 11, "Input/Output and the Operating System.")

6.14 Arguments to `main()`

Two arguments, conventionally called `argc` and `argv`, can be used with `main()` to communicate with the operating system. Here is a program that prints its command line arguments. It is a variant of the *echo* command in MS-DOS and UNIX.


```
/* Echoing the command line arguments. */

#include <stdio.h>

int main(int argc, char *argv[])
{
   int   i;

   printf("argc = %d\n", argc);
   for (i = 0; i < argc; ++i)
      printf("argv[%d] = %s\n", i, argv[i]);
   return 0;
}
```

The variable `argc` provides a count of the number of command line arguments. The array `argv` is an array of pointers to `char` that can be thought of as an array of strings. The strings are the words that make up the command line. Because the element `argv[0]` contains the name of the command itself, the value of `argc` is at least 1.

Suppose we compile the above program and put the executable code in the file *my_echo*. If we give the command

```
my_echo a is for apple
```

the following is printed on the screen:

```
argc = 5
argv[0] = my_echo
argv[1] = a
argv[2] = is
argv[3] = for
argv[4] = apple
```

On an MS-DOS system, the string in `argv[0]` consists of the full pathname of the command, and it is capitalized. As we will see in Section 11.12, "Environment Variables," on page 521, file names are often passed as arguments to `main()`.

6.15 Ragged Arrays

We want to contrast a two-dimensional array of type char with a one-dimensional array of pointers to char. Both similarities and differences exist between these two constructs.

```
#include <stdio.h>

int main(void)
{
   char   a[2][15] = {"abc:", "a is for apple"};
   char   *p[2]    = {"abc:", "a is for apple"};

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

The output of this program is the following:

```
abc abc: a is for apple
abc abc: a is for apple
```

The program and its output illustrate similarities in how the two constructs are used. Let us consider the program in some detail.

The identifier a is a two-dimensional array, and its declaration causes space for 30 chars to be allocated. The two-dimensional initializer is equivalent to

```
{{'a', 'b', 'c', ':', '\0'}, {'a', ' ', 'i', 's', ... }}
```

The identifier a is an array, each of whose elements is an array of 15 chars. Thus, a[0] and a[1] are arrays of 15 chars. Because arrays of characters are strings, a[0] and a[1] are strings. The array a[0] is initialized to

```
{'a', 'b', 'c', ':', '\0'}
```

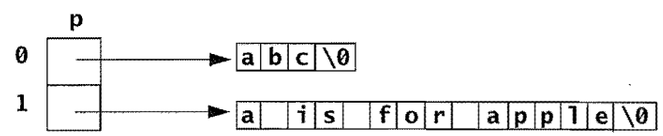
and because only five elements are specified, the rest are initialized to zero (the null character). Even though not all elements are used in this program, space has been allocated for them. The compiler uses a storage mapping function to access a[i][j]. Each access requires one multiplication and one addition.

The identifier p is a one-dimensional array of pointers to char. Its declaration causes space for two pointers to be allocated (4 bytes for each pointer on our machine). The element p[0] is initialized to point at "abc:", a string that requires space for 5 chars. The element p[1] is initialized to point at "a is ...", a string that requires space for 15 chars, including the null character \0 at the end of the string. Thus, p does its work in less space than a. Moreover, the compiler does not generate code for a storage mapping function to access p[i][j], which means that p does its work faster than a. Note that a[0][14] is a valid expression, but that p[0][14] is not. The expression p[0][14] overruns the bounds of the string pointed to by p[0]. Of course, a[0][14] overruns the string currently stored in a[0], but it does not overrun the array a[0]. Hence, the expression a[0][14] is acceptable.

Another difference is that the strings pointed to by p[0] and p[1] are constant strings, and, hence, cannot be changed. In contrast to this, the strings pointed to by a[0] and a[1] are modifiable.

An array of pointers whose elements are used to point to arrays of varying sizes is called a *ragged array*. Because, in the preceding program, the rows of p have different lengths, it is an example of a ragged array. If we think of the elements p[i][j] arranged as a "rectangular" collection of elements in rows and columns, the disparate row lengths give the "rectangle" a ragged look. Hence the name "ragged array."

A ragged array



6.16 Functions as Arguments

In C, pointers to functions can be passed as arguments, used in arrays, returned from functions, and so forth. In this section we describe how this facility works.

Suppose we want carry out a computation with a variety of functions. Consider

$$\sum_{k=m}^{n} f^2(k)$$

where in one instance $f(k)$ = `sin(k)`, and in another instance $f(k)$ = `1 / k`. The following routine accomplishes the task:

In file sum_sqr.c

```
#include "sum_sqr.h"

double sum_square(double f(double x), int m, int n)
{
   int      k;
   double   sum = 0.0;

   for (k = m; k <= n; ++k)
      sum += f(k) * f(k);
   return sum;
}
```

In the header to the function definition, the first parameter declaration tells the compiler that `f` is a function that takes an argument of type `double` and returns a `double`. The identifier `x` is for the human reader; the compiler disregards it. Thus, we could have written

```
double sum_square(double f(double), int m, int n)
{
   .....
```

When a function occurs in a parameter declaration, the compiler interprets it as a pointer. Here is an equivalent header to the function definition:

```
double sum_square(double (*f)(double), int m, int n)
{
   .....
```

We read the first parameter declaration as "`f` is a pointer to function that takes a single argument of type `double` and returns a `double`." The parentheses are necessary because `()` binds tighter than *. In contrast, consider the declaration

```
double   *g(double);
```

This declares `g` to be a function that takes an argument of type `double` and returns a pointer to `double`.

In the body of the function definition for `sum_square()`, we can either treat the pointer `f` as if it were a function, or we can explicitly dereference the pointer. For example, we could have written

```
sum += (*f)(k) * (*f)(k);        instead of        sum += f(k) * f(k);
```

It is helpful to think of the construct `(*f)(k)` as follows:

`f`	the pointer to a function
`*f`	the function itself
`(*f)(k)`	the call to the function

To illustrate how the function `sum_square()` might be used, let us write a complete program. In `main()`, we will use `sin()` from the mathematics library and the function `f()`, which we will write ourselves.

In file sum_sqr.h

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

double   f(double x);
double   sum_square(double f(double x), int m, int n);
```

In file main.c

```
#include "sum_sqr.h"

int main(void)
{
   printf("%s%.7f\n%s%.7f\n",
      " First computation: ", sum_square(f, 1, 10000),
      "Second computation: ", sum_square(sin, 2, 13));
   return 0;
}
```

In file fct.c

```
#include "sum_sqr.h"

double f(double x)
{
   return 1.0 / x;
}
```

The output of this program is

```
 First computation: 1.6448341
Second computation: 5.7577885
```

In mathematics it is known that the sum of $1/k^2$ from 1 to infinity is $\pi^2/6$. Notice that the first number in the output of the program approximates this.

Functions as Formal Parameters in Function Prototypes

There are a number of equivalent ways to write a function prototype that has a function as a formal parameter. To illustrate this, let us write a list of equivalent function prototypes for `sum_sqr()`:

```
double   sum_square(double f(double x), int m, int n);
double   sum_square(double f(double), int m, int n);
double   sum_square(double f(double), int, int);
double   sum_square(double (*f)(double), int, int);
double   sum_square(double (*)(double), int, int);
double   sum_square(double g(double y), int a, int b);
```

In the above list, the identifiers `m` and `n` are optional; if present, the compiler disregards them. In a similar fashion, the identifier `f` in the construct `(*f)` is optional.

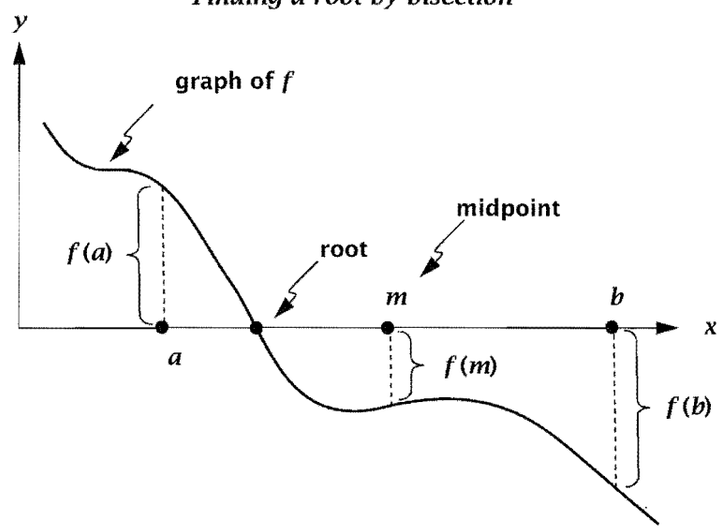
In C, a function can have multiple prototypes, provided that they are all equivalent. Thus, when we put the above list in the header file *sum_sqr.h*, our program compiles and runs just as before.

6.17 An Example: Using Bisection to Find the Root of a Function

An important problem that arises in engineering, mathematics, and physics is to find the root of a given real-valued function. A real number x that satisfies the equation $f(x) = 0$ is called a root of f. In simple cases—for example, if f is a quadratic polynomial—a formula for the roots is known. In general, however, there is no formula and a root must be found by numerical methods.

Suppose that f is a continuous real-valued function defined on the interval $[a, b]$. If $f(a)$ and $f(b)$ are of opposite sign, then the continuity of the function guarantees that it has a root in the interval $[a, b]$.

Finding a root by bisection



Notice that the condition that $f(a)$ and $f(b)$ have opposite sign is equivalent to the product $f(a) \times f(b)$ being negative. The method of bisection proceeds as follows. Let m be the midpoint of the interval. If $f(m)$ is zero, we have found a root. If not, then either $f(a)$ and $f(m)$ are of opposite sign or $f(m)$ and $f(b)$ are of opposite sign. Suppose that the first case holds. We then know that f has a root in the interval $[a, m]$, and we now start the process over again. After each iteration, we obtain an interval that is half the length of the previous interval. When the interval is sufficiently small, we take its midpoint as an approximation to a root of f. In general, we cannot hope to find the exact root. For most functions the precise mathematical root will not have an exact machine representation. Here is a program that finds an approximate root of the polynomial:

$$x^5 - 7x - 3$$

In file find_root.h

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

typedef   double   dbl;
```

```
extern int         cnt;
extern const dbl   eps;          /* epsilon, a small quantity */

dbl   bisection(dbl f(dbl x), dbl a, dbl b);
dbl   f(dbl x);
```

In file main.c

```
/* Find a root of f() by the bisection method. */

#include "find_root.h"

int         cnt = 0;
const dbl   eps = 1e-13;          /* epsilon, a small quantity */

int main(void)
{
   dbl   a = -10.0;
   dbl   b = +10.0;
   dbl   root;

   assert(f(a) * f(b) <= 0.0);
   root = bisection(f, a, b);              /* recursive fct call */
   printf("%s%d\n%s% .15f\n%s% .15f\n",
      "No. of fct calls: ", cnt,
      "Approximate root: ", root,
      "  Function value: ", f(root));
   return 0;
}
```

In file bisection.c

```
#include "find_root.h"

dbl bisection(dbl f(dbl x), dbl a, dbl b)
{
   dbl   m = (a + b) / 2.0;          /* midpoint */

   ++cnt;                            /* # of fct calls */
   if (f(m) == 0.0 || b - a < eps)
      return m;
   else if (f(a) * f(m) < 0.0)
      return bisection(f, a, m);
   else
      return bisection(f, m, b);
}
```


In file fct.c

```
#include "find_root.h"

dbl f(dbl x)
{
   return (x * x * x * x * x  -  7.0 * x  -  3.0);
}
```

In mathematics, the Greek letter "epsilon," written ε, is often used to represent a small positive quantity (number). In mathematical programming, the identifier `eps` is often used to play the role analogous to ε in mathematics.

In `main()`, `a` has value `-10` and `b` has value `+10`. Thus, the interval `[a, b]` is the same as the interval `[-10, +10]`. Clearly, `f(-10)` is negative and `f(+10)` is positive. That is, `f` takes on values having opposite signs at the end points of the interval `[a, b]`, which means that the bisection method can be used to find an approximate root of `f`. It so happens that `f` has three real roots and a pair of complex roots. The bisection method will only find a real root, and which one depends on the interval `[a, b]` initially chosen. Here is the output from the program:

```
No. of fct calls: 49
Approximate root:  1.719628091484431
  Function value: -0.000000000000977
```

If we use `1e-15` instead of `1e-13` as the value for `eps` in `main()`, then the function value printed out would be zero. We did not use `1e-15` in our program, because we did not want to give the reader the impression that an actual root was found. If we change our program to use `long doubles` instead of `doubles`, change the value of `eps` from `1e-13` to `1e-31`, and change the formats `% .16f` in the `printf()` statement to `% .33f`, then the following gets printed on our system:

```
No. of fct calls: 109
Approximate root:  1.719628091484457524492214890955915
  Function value: -0.000000000000000000000000000000555
```

No matter how much work we do, we do not expect to get an exact decimal representation of the root. Although we are not sure, the root is probably an irrational number.

The blank space following the % in the formats causes a blank to be printed if the corresponding argument is positive, but has no effect if the argument is negative. This provides a mechanism for lining up positive and negative numbers. (See Section 11.1, "The Output Function `printf()`," on page 493, for details about formats.)

The Kepler Equation

In the early 1600s Johannes Kepler wanted to solve the equation

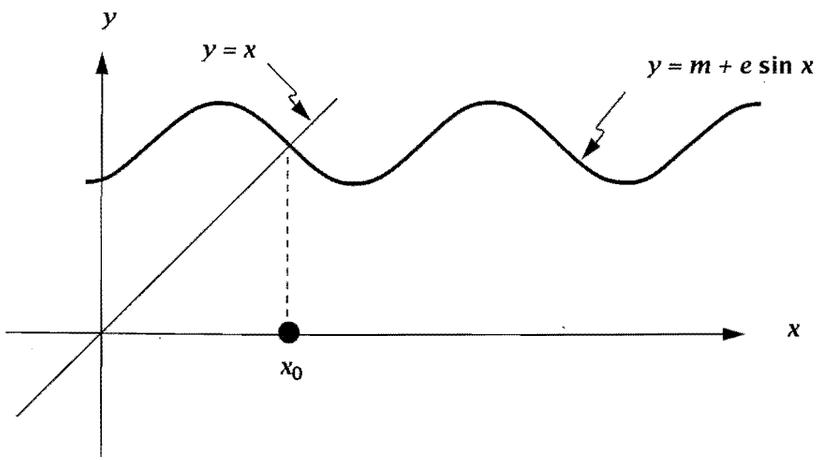
$$m = x - e \sin(x)$$

for various values of the parameters m and e. One way to view the problem is to graph

$$y = x \qquad \text{and} \qquad y = m + e \sin(x)$$

together. A solution to the Kepler equation then corresponds to the point where the two graphs intersect.

A solution of the Kepler equation



We want to write a program that solves the Kepler equation when the parameters have the values $m = 2.2$ and $e = 0.5$. Observe that solving

$$m = x - e \sin(x) \qquad \text{is equivalent to solving} \qquad x - e \sin(x) - m = 0$$

Our program will solve the Kepler equation by using the method of dissection to find a root of the function $x - e \sin(x) - m$. Here is our program:

In file kepler.h

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

typedef   double   dbl;

extern int         cnt;
extern const dbl   eps;          /* epsilon, a small quantity */
extern const dbl   e;          /* a parameter in the Kepler eqn */
extern const dbl   m;          /* a parameter in the Kepler eqn */

dbl   bisection(dbl f(dbl x), dbl a, dbl b);
dbl   kepler(dbl x);
```

In file main.c

```
/* Use bisection to solve the Kepler equation. */

#include "kepler.h"

int         cnt = 0;
const dbl   eps = 1e-15;          /* epsilon, a small quantity */
const dbl   e   = 0.5;          /* a parameter in the Kepler eqn */
const dbl   m   = 2.2;          /* a parameter in the Kepler eqn */

int main(void)
{
   dbl   a = -100.0;
   dbl   b = +100.0;
   dbl   root;

   assert(kepler(a) * kepler(b) <= 0.0);
   root = bisection(kepler, a, b);        /* recursive fct call */
   printf("%s%d\n%s% .15f\n%s% .15f\n",
      "No. of fct calls: ", cnt,
      "Approximate root: ", root,
      "  Function value: ", kepler(root));
   return 0;
}
```

In file kepler.c

```
#include "kepler.h"

dbl kepler(dbl x)
{
   return (x - e * sin(x) - m);
}
```

The `bisection()` function is the same as the one we wrote Section 6.17, "An Example: Using Bisection to Find the Root of a Function," on page 298. Here is the output of the program:

```
No. of fct calls: 59
Approximate root:  2.499454528163501
  Function value:  0.000000000000000
```

6.18 Arrays of Pointers to Function

In the *kepler* program that we discussed in the previous section, the `bisection()` function takes as its first argument a function, or more precisely, a pointer to function. In C, a function name by itself is treated by the compiler as a pointer to the function. (This is analogous to the idea that an array name by itself is treated as a pointer to the base of the array in memory.) Technically speaking, when we pass `kepler` as an argument to `bisection()`, it is the address of `kepler()` that gets passed. To make the pointer nature of the identifier `kepler` more evident, here is another way to write `main()` in the *kepler* program:

```
int main(void)
{
   dbl   a = -100.0;
   dbl   b = +100.0;
   dbl   root;
   dbl   (*pfdd)(dbl);                  /* ptr to fct taking dbl
                                           and returning dbl     */
   pfdd = kepler;
   assert(pfdd(a) * pfdd(b) <= 0.0);
   root = bisection(pfdd, a, b);        /* recursive fct call */
   .....
```


Observe that `pfdd` is a pointer variable of type "pointer to function taking a single argument of type `double` and returning a `double`." Like other C pointers, `pfdd` is strongly typed and can only be assigned pointer values of the same type. Because `kepler()` is a function taking a `double` and returning a `double`, we can assign `kepler` to `pfdd`. Because the type for `pfdd` matches the type of the first parameter in the function definition for `bisection()`, we can pass `pfdd` as an argument to `bisection()`.

In our next program we create an array of pointers, each of type "pointer to function taking a `double` and returning a `double`." We then deal with each array element (function pointer) in a `for` loop. *Caution:* In this program `pfdd` will be a type, not a pointer variable.

In file find_roots.h

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

#define   N   4                  /* size of array of ptrs to fcts */

typedef   double   dbl;

/*
// Create the type "ptr to fct taking a dbl and returning a dbl."
*/
typedef   dbl   (*pfdd)(dbl);

extern int         cnt;
extern const dbl   eps;            /* epsilon, a small quantity */

dbl   bisection(pfdd f, dbl a, dbl b);
dbl   f1(dbl x);
dbl   f2(dbl x);
dbl   f3(dbl x);
```

In file main.c

```
/* Use bisection to find roots. */

#include "find_roots.h"

int         cnt = 0;
const dbl   eps = 1e-13;          /* epsilon, a small quantity */
int main(void)
{
   int    begin_cnt;
   int    i;
   int    nfct_calls;
   dbl    a = -100.0;
   dbl    b = +100.0;
   dbl    root;
   dbl    val;
   pfdd   f[N] = {NULL, f1, f2, f3};

   for (i = 1; i < N; ++i) {
      assert(f[i](a) * f[i](b) <= 0.0);
      begin_cnt = cnt;
      root = bisection(f[i], a, b);
      nfct_calls = cnt - begin_cnt;
      val = f[i](root);
      printf("%s%d%s% .15f\n%s%d%s% .15f\n%s%3d\n\n",
         "For f[", i, "](x) an approximate root is x0 = ", root,
         " Fct evaluation at the root: f[", i, "](x0) = ", val,
         "   Number of fct calls to bisection() =", nfct_calls);
   }
   return 0;
}
```

In file bisection.c

```
#include "find_roots.h"

dbl bisection(pfdd f, dbl a, dbl b)
{
   dbl   m = (a + b) / 2.0;          /* midpoint */

   ++cnt;                            /* # of fct calls */
   if (f(m) == 0.0 || b - a < eps)
      return m;
   else if (f(a) * f(m) < 0.0)
      return bisection(f, a, m);
   else
      return bisection(f, m, b);
}
```


In file fct.c

```
#include "find_roots.h"

dbl f1(dbl x)
{
   return (x*x*x - x*x + 2.0*x - 2.0);
}

dbl f2(dbl x)
{
   return (sin(x) - 0.7*x*x*x + 3.0);
}

dbl f3(dbl x)
{
   return (exp(0.13*x) - x*x*x);
}
```

When we execute this program, here is what gets printed:

```
For f[1](x) an approximate root is x0 =  1.000000000000023
 Fct evaluation at the root: f[1](x0) =  0.000000000000069
   Number of fct calls to bisection() = 52

For f[2](x) an approximate root is x0 =  1.784414278382185
 Fct evaluation at the root: f[2](x0) =  0.000000000000169
   Number of fct calls to bisection() = 52

For f[3](x) an approximate root is x0 =  1.046387173807117
 Fct evaluation at the root: f[3](x0) = -0.000000000000134
   Number of fct calls to bisection() = 52
```

Dissection of the *find_roots* Program

```
/*
// Create the type "ptr to fct taking a dbl and returning a dbl."
*/
typedef   dbl   (*pfdd)(dbl);
```

This typedef makes pfdd a new name for the type "pointer to function taking a single argument of type double and returning a double." We will see that the use of pfdd makes the code easier to write and easier to read.

```
dbl   bisection(pfdd f, dbl a, dbl b);
```

This is the function prototype for `bisection()`. Because `pfdd` is a `typedef`, the parameter declaration `pfdd f` is equivalent to what one gets by substituting `f` for `pfdd` in the `typedef` itself. Thus, `pfdd f` is equivalent to `dbl (*f)(dbl)`, and two other equivalent function prototypes for `bisection()` are

```
dbl   bisection(dbl f(dbl), dbl a, dbl b);
dbl   bisection(dbl f(dbl x), dbl a, dbl b);
```

Although these are longer, some programmers prefer them. Any of these styles is acceptable.

```
pfdd   f[N] = {NULL, f1, f2, f3};
```

We declare `f` to be an array of `N` elements, with each element having type `pfdd`. The array is initialized with four pointer values, the first being `NULL`. The initialization causes `f[0]` to be assigned `NULL`, `f[1]` to be assigned to `f1`, `f[2]` assigned to `f2`, and so forth. We will not use `f[0]`. The following declarations are equivalent:

```
pfdd   f[N] = {NULL, f1, f2, f3};
pfdd   f[N] = {NULL, &f1, &f2, &f3};
dbl    (*f[N])(dbl) = {NULL, f1, f2, f3};
dbl    (*f[N])(dbl) = {NULL, &f1, &f2, &f3};
```

We can choose any one from this list to use as our declaration of `f`. A function name by itself, such as `f1`, can be thought of as a pointer. But we may also think of it as the name of a function with `&f1` being a pointer to the function. ANSI C allows ambiguous thinking in this matter.

```
root = bisection(f[i], a, b);
```

Here, the pointer `f[i]` is being passed as an argument to `bisection()`. Equivalently, we could have written

```
root = bisection(*f[i], a, b);
```

When we dereference the pointer, we get the function that it points to. But the compiler treats a function by itself as a pointer to the function, so the two ways of writing this are equivalent. The words "function by itself" mean that the function is not followed by parentheses. If a function name or a function pointer or a dereferenced function pointer is followed by parentheses, then we have a call to the function.


```
f[i](a)        f[i](b)        f[i](root)
```

These are calls to the function pointed to by `f[i]`. If we wish, we can write

```
(*f[i])(a)        (*f[i])(b)        (*f[i])(root)
```

instead. The output shows that `f[i](root)` for each `i` is close to zero, as it should be.

Note that each of our functions `f1()`, `f2()`, and `f3()` happens to take on values with opposite signs at the end points of the interval [-100, 100]. Our calls to `bisection()` would not work if this condition does not hold.

6.19 The Type Qualifiers const and volatile

The keywords `const` and `volatile` have been added to the C language by the ANSI C committee. These keywords are not available in traditional C. They are called *type qualifiers* because they restrict, or qualify, the way an identifier of a given type can be used.

Let us first discuss how `const` is used. Typically, in a declaration `const` comes after the storage class, if any, but before the type. Consider the declaration

```
static const int   k = 3;
```

We read this as "`k` is a constant `int` with `static` storage class." Because the type for `k` has been qualified by `const`, we can initialize `k`, but thereafter `k` cannot be assigned to, incremented, decremented, or otherwise modified.

In C, even though a variable has been qualified with `const`, it still cannot be used to specify an array size in another declaration. In C++, however, it can be used for this purpose. This is one of the places where C and C++ differ.

```
const int   n = 3;
int         v[n];        /* any C compiler should complain */
```

In some situations we can use a `const`-qualified variable instead of a symbolic constant; in other situations, we cannot.

An unqualified pointer should not be assigned the address of a `const`-qualified variable. The following code illustrates the problem:

```
const int   a = 7;
int         *p = &a;     /* the compiler will complain */
```

Here is the reason why the compiler complains. Since p is an ordinary pointer to `int`, we could use it later in an expression such as ++*p to change the stored value of a, violating the concept that a is constant. If, however, we write

```
const int   a = 7;
const int   *p = &a;
```

then the compiler will be happy. The last declaration is read "p is a pointer to a constant `int` and its initial value is the address of a." Note that p itself is not constant. We can assign to it some other address. We may not, however, assign a value to *p. The object pointed to by p should not be modified.

Suppose we want p itself to be constant, but not a. This is achieved with the following declarations:

```
int           a;
int * const   p = &a;
```

We read the last declaration as "p is a constant pointer to `int`, and its initial value is the address of a." Thereafter, we may not assign a value to p, but we may assign a value to *p. Now consider

```
const int           a = 7;
const int * const   p = &a;
```

The last declaration tells the compiler that p is a constant pointer to a constant `int`. Neither p nor *p can be assigned to, incremented, or decremented.

In contrast to `const`, the type qualifier `volatile` is seldom used. A `volatile` object is one that can be modified in some unspecified way by the hardware. Now consider the declaration

```
extern const volatile int   real_time_clock;
```

The `extern` means "look for it elsewhere, either in this file or in some other file." The qualifier `volatile` indicates that the object may be acted on by the hardware. Because `const` is also a qualifier, the object may not be assigned to, incremented, or decremented within the program. The hardware can change the clock, but the code cannot.

Summary

1 The brackets [] are used in a declaration to tell the compiler that an identifier is an array. The integral constant expression in the brackets specifies the size of the array. For example, the declaration

```
int   a[100];
```

causes the compiler to allocate contiguous space in memory for 100 `int`s. The elements of the array are numbered from 0 to 99. The array name a by itself is a constant pointer; its value is the base address of the array.

2 A pointer variable takes addresses as values. Some typical values are `NULL`, addresses of variables, string constants, and pointer values, or addresses, returned from functions such as `calloc()`. If allocation fails—for example, the system free store (heap) is exhausted—then `NULL` is returned.

3 The address operator & and the indirection or dereferencing operator * are unary operators with the same precedence and right to left associativity as other unary operators. If v is a variable, then the expression

```
*&v          is equivalent to          v
```

4 Pointers are used as formal parameters in headers to function definitions to effect "call-by-reference." When addresses of variables are passed as arguments, they can be dereferenced in the body of the function to change the values of variables in the calling environment.

5 In C, arrays and pointers are closely related topics. If a is an array and i is an `int`, then the expression

```
a[i]          is equivalent to          *(a + i)
```

These expressions can be used to access elements of the array. The expression `a + i` is an example of pointer arithmetic. Its value is the address of the element of the array that is i elements beyond a itself. That is, `a + i` is equivalent to `&a[i]`.

6 In the header to a function definition, the declaration of a parameter as an array is equivalent to its declaration as a pointer. For example,

```
int a[]        is equivalent to        int *a
```

This equivalence does *not* hold elsewhere.

7 When an array is passed as an argument to a function, a pointer is actually passed. The array elements themselves are not copied.

8 Strings are one-dimensional arrays of characters. By convention, they are terminated with the null character `\0`, which acts as the end-of-string sentinel.

9 The standard library contains many useful string-handling functions. For example, `strlen()` returns the length of a string, and `strcat()` concatenates two strings.

10 Arrays of any type can be created, including arrays of arrays. For example,

```
double   a[3][7];
```

declares a to be an array of "array of 7 `double`s." The elements of `a` are accessed by expressions such as `a[i][j]`. The base address of the array is `&a[0][0]`, not `a`. The array name `a` by itself is equivalent to `&a[0]`.

11 In the header to a function definition, the declaration of a multidimensional array must have all sizes specified except the first. This allows the compiler to generate the correct storage mapping function.

12 Arguments to `main()` are typically called `argc` and `argv`. The value of `argc` is the number of command line arguments. The elements of the array `argv` are addresses of the command line arguments. We can think of `argv` as an array of strings.

13 Ragged arrays are constructed from arrays of pointers. The elements of the array can point to arrays with different sizes.

14 Like an array name, a function name that is passed as an argument is treated as a pointer. In the body of the function the pointer can be used to call the function in the normal way, or it can be explicitly dereferenced.

15 The type qualifiers `const` and `volatile` have been added to ANSI C. They are not available in traditional C.

Exercises

1 Four values get printed when the following code is executed. How many of those values are the same? Explain.

```
char   *format = "%p %d %d %d\n";
int    i = 3;
int    *p = &i;

printf(format, p, *p + 7, 3 * **&p + 1, 5 * (p - (p - 2)));
```

2 One of our compilers warned us about integer overflow for the expression `p - (p - 2)` in the previous exercise. Modify the program that you wrote in the previous exercise so that it prints the integer values of both `p` and `p - 2`. Does it seem possible that integer overflow can occur? (See the next exercise for further discussion.)

3 Consider the following program:

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

int main(void)
{
   int         a, b, *p = &a, *q = &b;
   ptrdiff_t   diff = p - q;
   printf("diff = %d\n", diff);
   return 0;
}
```

In ANSI C, the difference of two pointer expressions must be a signed integral type. On most UNIX systems, the type is `int`, and on most MS-DOS systems the type is `long`. On all ANSI C systems, the type is given in the standard header file *stddef.h* by a type definition of the following form:

```
typedef   type  ptrdiff_t;
```

Find this `typedef` in *stddef.h* on your system so that you will know the type for `diff`. Note that `%d` is appropriate in the `printf()` statement if `diff` has type `int` and that `%ld` is appropriate if `diff` has type `long`. Run the program so that you understand its effects. Then modify the program by adding the following two lines:

```
diff = p - (int *) 0;
printf("diff = %d\n", diff);
```

Are you surprised by what gets printed now? Do you understand the reason for the compiler warning that was discussed in exercise 2, on page 311? Explain. If `int *` is replaced by `ptrdiff_t *`, does the program act any differently?

4 If `i` and `j` are `int`s and `p` and `q` are pointers to `int`, which of the following assignment expressions are not legal?

```
p = &i          p = &*&i      i = (int) p      q = &p *q = &j
i = (*&)j       i = *&*&j     i = *p++ + *q
```

5 When variables are declared, are they located in memory contiguously? Write a program with the declaration

```
char   a, b, c, *p, *q, *r;
```

and print out the locations that are assigned to all these variables by your compiler. Are the locations in order? If the locations are in order, are they increasing or decreasing? Is the address of each pointer variable divisible by 4? If so, this probably means that each pointer value gets stored in a machine word.

6 The following program uses `%p` formats to print out some addresses:

```
#include <stdio.h>

int main(void)
{
   int   a = 1, b = 2, c = 3;

   printf("%s%p\n%s%p\n%s%p\n",
      "&a = ", &a,
      "&b = ", &b,
      "&c = ", &c);
   return 0;
}
```

If the variables `a`, `b`, and `c` are not initialized, does the program produce the same output? What happens on your system if you change `%p` to `%d`? Does your compiler complain? (It should.) If possible, run your program on an MS-DOS system. Because a pointer is 4 bytes and an `int` is 2 bytes, the `%d` format is inappropriate and can cause a negative number to be printed.

7 If you want to see addresses printed as decimal numbers rather than hexadecimals, it is usually safe to cast an address as an `unsigned long` and use the `%lu` format. Try this on your system by replacing the `printf()` statement in exercise 6, on page 312, by

```
printf("%s%lu\n%s%lu\n%s%lu\n",
   "&a = ", (unsigned long) &a,
   "&b = ", (unsigned long) &b,
   "&c = ", (unsigned long) &c);
```

8 What gets printed? Explain.

```
#include <stdio.h>

typedef   unsigned long   ulong;

int main(void)
{
   char          *pc = NULL;
   int           *pi = NULL;
   double        *pd = NULL;
   long double   *pld = NULL;

   printf("%5lu%5lu\n%5lu%5lu\n%5lu%5lu\n",
      (ulong)(pc + 1), (ulong)(pi + 1),
      (ulong)(pd + 1), (ulong)(pld + 1),
      (ulong)(pc + 3), (ulong)(pld + 3));
   return 0;
}
```

9 The following array declarations have several errors. Identify each of them.

```
#define   N   4

int   a[N] = {0, 2, 2, 3, 4};
int   b[N - 5];
int   c[3.0];
```

10 In the following program the invocation of `change_it()` seems to have no effect. Explain.

```
#include <stdio.h>

void   change_it(int []);

int main(void)
{
   int   a[5], *p;
   p = a;
   printf("p has the value %p\n", p);
   change_it(a);
   p = a;
   printf("p has the value %p\n", p);
   return 0;
}

void change_it(int a[])
{
   int   i = 777, *q = &i;
   a = q;     /* a is assigned a different value */
}
```

11 What is wrong with the following program? Correct the program and explain the meaning of its output.

```
#include <stdio.h>

int main(void)
{
   int   a[] = {0, 2, 4, 6, 8},
       *p = a + 3;

   printf("%s%d%s\n%s%d%s\n",
      "a[?]   = ", *p,      "?",
      "a[?+1] = ", *p + 1, "?");
   return 0;
}
```

12 A real polynomial $p(x)$ of degree n or less is given by

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

with the coefficients $a_0, a_1, \ldots, a_n$ representing real numbers. If `an != 0`, then the degree of $p(x)$ is n. Polynomials can be represented in a machine by an array such as


```
#define   N   5          /* N is the max degree */

double   p[N + 1];
```

Write a function

```
double eval(double p[], double x, int n)   /* n is max degree */
{
   .....
```

that returns the value of the polynomial p evaluated at x. Write two versions of the function. The first version should be written with a straightforward naive approach. The second version should incorporate Horner's Rule. For fifth-degree polynomials, Horner's Rule is expressed by writing

$$p(x) = a_0 + x(a_1 + (a_2 + x(a_3 + x(a_4 + x(a_5))))$$

How many additions and multiplications are used in each of your two versions of the `eval()` function?

13 Write a function that adds two polynomials of at most degree n.

```
/* f = g + h;  n is the max degree of f, g, and h */

void add(double f[], double g[], double h[], int n)
{
   .....
```

14 Write an algorithm to multiply two polynomials of at most degree n. Use your function `add()` to sum intermediate results. This is not very efficient. Can you write a better routine?

15 Modify the function `bubble()` so that it terminates after the first pass in which no two elements are interchanged.

16 Modify `mergesort()` so that it can be used with an array of any size, not just with a size that is a power of 2. Recall that any positive integer can be expressed as a sum of powers of two—for example,

$$27 = 16 + 8 + 2 + 1$$

Consider the array as a collection of subarrays of sizes that are powers of 2. Sort the subarrays, and then use `merge()` to produce the final sorted array.

17 If p is a pointer, then the two expressions *p++ and (*p)++ have different effects. With the following code, what gets printed? Explain.

```
char   a[] = "abc";
char   *p;
int    i;

p = a;
for (i = 0; i < 3; ++i)
   printf("%c\n", *p++);
printf("a = %s\n", a);
p = a;
for (i = 0; i < 3; ++i)
   printf("%c\n", (*p)++);
printf("a = %s\n", a);
```

18 Look carefully at the function definition for `strcpy()` given in Section 6.11, "String-Handling Functions in the Standard Library," on page 273, to see that it makes sense to copy the tail end of a string onto its beginning. With the following code, what gets printed? Explain.

```
char   a[] = "abcdefghijklmnopqrstuvwxyz";
char   *p = a;
char   *q = a + strlen(a) - 3;

printf("a = %s\n", a);
strcpy(p, q);
printf("a = %s\n", a);
```

Explain what would go wrong if we were to interchange p and q in the call to `strcpy()`.

19 Write a program that will test the relative efficiency of the function `bubble()` given in Section 6.7, "An Example: Bubble Sort," on page 257, versus the function `mergesort()` that you wrote in the previous exercise. Generate test data by using `rand()` to fill arrays. Run your program on arrays of various sizes, say with 10, 100, 500, and 1,000 elements. Plot the running time for each sort versus the size of the array. For large array sizes, you should see the growth indicated by the formulas given in the text. For small array sizes, there is too much overhead to detect this growth pattern. If you are unfamiliar with how to time program execution, see Section 11.16, "How to Time C Code," on page 528. If you are on a UNIX system, you can give the following command to time a program:

time pgm

20 A palindrome is a string that reads the same both forward and backward. Some examples are

```
"ABCBA"     "123343321"     "otto"     "i am ma i"     "C"
```

Write a function that takes a string as an argument and returns the `int` value 1 if the string is a palindrome and returns 0 otherwise. If UNIX is available to you, how many palindromes can you find in the file */usr/dict/words*?

21 Modify your palindrome function from the previous so that blanks and capitals are ignored in the matching process. Under these rules, the following are examples of palindromes:

```
"Anna"     "A man a plan a canal Panama"     "ott  o"
```

If UNIX is available to you, how many more palindromes can you find in the file */usr/dict/words*?

22 What gets printed? Explain.

```
printf("%c%c%c%c%c!\n",
   "ghi"[1], *("def" + 1),
   *"abc" + 11, "klm"[1], *"ghi" + 8);
```

23 In Section 3.4, "The Data Type `int`," on page 116, we saw that the largest value that can be stored in a `long int` is approximately 2 billion. In many applications, such numbers are not big enough. For example, the federal government has to deal with figures in the trillions of dollars. (Or is it quadrillions?) In this exercise, we want to explore, in a primitive fashion, how two large integers can be added. Here is a program that will do this:

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

#define   N   20                        /* size of all arrays */

typedef   const char   cchr;

void   add(int sum[], int a[], int b[]);     /* sum = a + b */
void   wrt(cchr *s, int a[]);
```

```
int main(void)
{
   int   a[N]    = {7, 5, 9, 8, 9, 7, 5, 0, 0, 9, 9, 0, 8, 8};
   int   b[N]    = {7, 7, 5, 3, 1, 2, 8, 8, 9, 6, 7, 7};
   int   sum[N];
   int   ndigits;

   add(sum, a, b);
   wrt("Integer a: ", a);
   wrt("Integer b: ", b);
   wrt("      Sum: ", sum);
   return 0;
}

void add(int sum[], int a[], int b[])
{
   int   carry = 0;
   int   i;

   for (i = 0; i < N; ++i) {
      sum[i] = a[i] + b[i] + carry;
      if (sum[i] < 10)
         carry = 0;
      else {
         carry = sum[i] / 10;
         sum[i] %= 10;
      }
   }
}

void wrt(cchr *s, int a[])
{
   int   i;

   printf("%s", s);
   /*
   // Print leading zeros as blanks.
   */
   for (i = N - 1; i > 0 && a[i] == 0; --i)
         putchar(' ');
   /*
   // After a leading digit greater than zero is found,
   // print all the remaining digits, including zeros.
   */
   for ( ; i >= 0; --i)
      printf("%d", a[i]);
   putchar('\n');
}
```


When we execute this program, here is what appears on the screen:

```
Integer a:       88099005798957
Integer b:         776988213577
      Sum:       88875994012534
```

Note that the digits are stored in array elements going from element 0 to element `N - 1`, but that the digits are printed in the opposite order. To understand this program, review how you learned to do addition in grade school. Write a similar program that computes the product of two integers.

24 The `sizeof` operator can be used to find the number of bytes needed to store a type or an expression. When applied to arrays, it does *not* yield the size of the array. What gets printed? Explain.

```
#include <stdio.h>

void   f(int a[]);

int main(void)
{
   char     s[] = "deep in the heart of texas";
   char     *p  = "deep in the heart of texas";
   int      a[3];
   double   d[5];

   printf("%s%d\n%s%d\n%s%d\n%s%d\n",
      "sizeof(s) = ", sizeof(s),
      "sizeof(p) = ", sizeof(p),
      "sizeof(a) = ", sizeof(a),
      "sizeof(d) = ", sizeof(d));
   f(a);
   return 0;
}

void f(int a[])
{
   printf("In f(): sizeof(a) = %d\n", sizeof(a));
}
```

25 If UNIX is available to you and you are familiar with the *diff* utility, try the following experiment. Use the two statements

```
printf("abc\n");    and    printf("a%cb%cc\n", '\0', '\0');
```

to write two versions of an elementary program that writes on the screen. Use redirection to put the printout of the respective programs into two files, say *tmp1* and

tmp2. If you use the UNIX utility *cat* to print first one file on the screen and then the other, you will not see any difference. Now try the command

```
diff tmp1 tmp2
```

Do you see what the confusion is? Explain. *Hint:* Use the *od* command with the -c option to get a more complete view of what is in the files. By the way, why did we use the %c format? Why not just print the string "a\0b\0c\n"?

26 In traditional C, changing the contents of a string constant was allowed, although it was considered poor programming practice to do so. In ANSI C, the programmer is not supposed to be able to change a string constant. However, compilers vary in their ability to enforce this. Consider the following code:

```
char   *p = "abc";

*p = 'X';                  /* illegal? */
printf("%s\n", p);         /* Xbc gets printed? */
```

On our system, one of our compilers does not complain and the program executes, whereas another compiler exhibits a run-time error. What happens on your system?

27 Consider the following code:

```
char   *p = "abc", *q = "abc";

if (p == q)
   printf("The two strings have the same address!\n");
else
   printf("As I expected, the addresses are different.\n");
```

Both p and q have been initialized to the base address in memory of a string constant—namely, "abc". Note that p == q tests whether two pointer values are the same; it is *not* a test for equality of the contents of the strings pointed to by p and q. Are there two string constants in memory or only one? This is compiler-dependent. Moreover, many compilers provide an option that determines whether all string constants with the same content get stored separately or as just one string. In traditional C, because string constants could be overwritten, string constants with the same content were usually stored separately. (See exercise 26, on page 320.) In contrast to this, many ANSI C compilers store them in the same place. What happens on your system?

28 The ANSI C committee has introduced the type qualifier const as a new keyword in the C language. Here is an example of its use:

```
const char   *p;
```

Here, the type qualifier const tells the compiler that the character in memory pointed to by p should not be changed. (Read "p is a pointer to a constant char.") Compilers vary on their ability to enforce this. Try the following code:

```
char         s[] = "abc";
const char   *p = s;

*p = 'A';                  /* illegal? */
printf("%s\n", s);
```

Does your compiler complain? (It should.)

29 A lot of effort has been expended on the problem of machine translation. How successful is a naive approach? Go to a library to find out what the most common, say 100, English words are. Consult, for example, *The American Heritage Word Frequency Book* by John Carroll et al. (Boston, MA: Houghton Mifflin, 1971). Write down the 100 words and, with the help of a foreign language dictionary, write down their translation. Write a program that uses two arrays such as

```
char   *foreign[100], *english[100];
```

to translate foreign text to English. Test your program. (You may be surprised at the results.) Instead of 100 words, try 200. Does your program produce a significantly better translation?

30 A simple encryption scheme is to interchange letters of the alphabet on a one-to-one basis. This can be accomplished with a translation table for the 52 lower- and uppercase letters. Write a program that uses such a scheme to encode text. Write another program that will decode text that has been encoded. This is not a serious encryption scheme. Do you know why? If you are interested, learn about a more secure encryption system and then program it. If UNIX is available to you, read the on-line manual concerning *crypt* to get the flavor of some of the concerns in the area of encryption.

31 What gets printed? Explain.

```
#include <stdio.h>

void   try_me(int [][3]);

int main(void)
{
   int   a[3][3] = {{2, 5, 7}, {0, -1, -2}, {7, 9, 3}};

   try_me(a);
   return 0;
}

void try_me(int (*a)[3])
{
   printf("%d %d %d %d  . . .  infinity\n",
      a[1][0], -a[1][1], a[0][0], a[2][2]);
}
```

Now, change the declaration of the parameter in the header of the function definition of `try_me()` to

```
int *a[3]
```

and leave the rest of the code alone. Does your compiler complain? (It should.) Explain.

32 Choose a character and use a two-dimensional array that matches the size of your screen to graph on the screen the functions `sin()` and `cos()` from 0 to 2π. Because, on most screens, the space in which a character is printed is not square, there is horizontal/vertical distortion. Experiment with your graphs to see if you can remove this distortion.

33 Write out a dissection for the following program. An understanding of the storage mapping function is needed to explain it. A complete explanation of the last `printf()` statement is rather technical and should be attempted only by advanced computer science students.


```
#include <stdio.h>

int main(void)
{
   int   a[3][5], i, j,
       *p = *a;     /* a nice initialization! */

   for (i = 0; i < 3; ++i)
      for (j = 0; j < 5; ++j)
         a[i][j] = i * 5 + j;
   for (i = 0; i < 3; ++i)
      for (j = 0; j < 5; ++j)
         printf("%s%12d", (j == 0) ? "\n" : "", a[i][j]);
   printf("\n");
   for (i = 0; i < 15; ++i)
         printf("%s%12d", (i % 5 == 0) ? "\n" : "", *(p + i));
   printf("\n\n%12d%12d\n%12d%12d\n%12d%12d\n%12d%12d\n\n",
      **a, **(a + 1),
      *(a[0] + 1), *(*a + 1),
      *(a[1] + 2), *(*(a + 1) + 2),
      *(a[2] + 3), *(*(a + 2) + 3));
   printf("%-11s%s%12d\n%-11s%s%12d\n%-11s%s%12d\n\n",
      "(int) a", "=", (int) a,
      "(int) *a", "=", (int) *a,
      "(int) **a", "=", (int) **a);
   return 0;
}
```

34 Modify the *my_echo* program in Section 6.14, "Arguments to `main()`," on page 291, so that it will print out its arguments in capital letters if the option *-c* is present. Do not print out the argument that contains the option.

35 Complete the following table:

Declarations and initialization		
`char *p[2][3] = { "abc", "defg", "hi", "jklmno", "pqrstuvw", "xyz" };`		
Expression	**Equivalent expression**	**Value**
`***p`	`p[0][0][0]`	`'a'`
`**p[1]`		
`**(p[1] + 2)`		
`*(*(p + 1) + 1)[7]`	`/* error */`	
`(*(*(p + 1) + 1))[7]`		
`*(p[1][2] + 2)`		

36 Simulations that involve the repeated use of a random-number generator to reproduce a probabilistic event are called Monte Carlo simulations, so called because Monte Carlo has one of the world's most famous gaming casinos. In this exercise we want to find the probability that at least two people in a room with n people have birthdays that fall on the same day of the year. Assume that there are 365 days in a year, and assume that the chance of a person being born on each day of the year is the same. A single trial experiment consists of filling an array of size n with integers that are randomly distributed from 1 to 365. If any two elements in the array have the same value, then we say that the trial is "true." Thus, a true trial corresponds to the case when at least two people in the room were born on the same day of the year. Simulate the probability by running, say, 10,000 trials with n people in the room. Do this for $n = 2, 3, \ldots, 100$. You can use the expression

```
rand() % 365 + 1
```

to compute the day of birth for each person. (Use a better random-number generator, such as `lrand48()`, if one is available to you.) The number of true trials divided by 10,000 is the computed simulated probability. What value of n yields the break-even point? That is, find the least n for which the probability is 1/2 or more.

37 Modify the function `merge()` given in the text and the function `mergesort()` that you wrote in exercise 16, on page 315, to work on arrays of pointers to `char` rather than on arrays of `int`s. Modify the *sort_words* program to use these functions. Time both versions of the program on a large file. The program that uses `mergesort()` should run much faster. Does it?

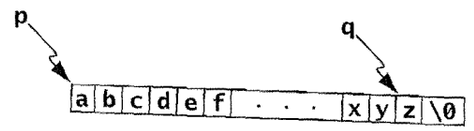
38 The following code can be used to reverse the characters in a string:

```
char *reverse(char *s)
{
   char   *p, *q, tmp;
   int    n;

   n = strlen(s);
   q = (n > 0) ? s + n - 1 : s;
   for (p = s; p < q; ++p, --q) {
      tmp = *p;
      *p = *q;
      *q = tmp;
   }
   return s;
}
```

Test this function by writing a program that contains the following lines:

```
char   str[] = "abcdefghijklmnopqrstuvwxyz";
printf("%s\n", reverse(str));
```

Execute your program so that you understand its effects. The following shows what is in memory at the beginning of the `for` loop:



Explain why this picture is correct. Draw a series of similar pictures that describe the state of the memory after each iteration of the `for` loop.

39 Remember that `argc` and `argv` are typically used as arguments to `main()`. Because `argc` is the number of command line arguments, one might think that the size of the array `argv` is `argc`, but that is not so. The array `argv` has size `argc + 1`, and the last element in the array is the null pointer. Here is another version of the *echo* command:

```
#include <stdio.h>

int main(int argc, char **argv)
{
   while (*argv != NULL)
      printf("%s  ", *argv++);
   putchar('\n');
   return 0;
}
```

Run the program so that you understand its effects. Give a detailed explanation of how it works. *Hint:* If a pointer to pointer to `char` is dereferenced, then the result is a pointer to `char`. Also, reread the dissection of the `strcat()` function in Section 6.11, "String-Handling Functions in the Standard Library," on page 276.

40 In C, a function prototype can occur more than once. Moreover, equivalent function prototypes do not conflict. Modify the *find_roots* program that we wrote in Section 6.18, "Arrays of Pointers to Function," on page 302, by replacing the function prototype

```
dbl   bisection(dbl f(dbl x), dbl a, dbl b);
```

with the following list of function prototypes:

```
dbl   bisection(pfdd, dbl, dbl);
dbl   bisection(pfdd f, dbl a, dbl b);
dbl   bisection(dbl (*)(dbl), dbl a, dbl b);
dbl   bisection(dbl (*f)(dbl), dbl a, dbl b);
dbl   bisection(dbl f(dbl), dbl a, dbl b);
dbl   bisection(dbl f(dbl x), dbl a, dbl b);
```

Does your compiler complain? (It shouldn't.)

41 In Section 6.18, "Arrays of Pointers to Function," on page 302, we saw that the *find_roots* program made 49 calls to `root()` each time a root of a function was computed. Use a hand calculator to explain why precisely 49 calls were made. *Hint:* Most of the calls to `root()` cut the interval in half.

42 Compile and execute the following program so that you understand its effects:

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

void   tell_me(int f(const char *, const char *));

int main(void)
{
   tell_me(strcmp);
   tell_me(main);
   return 0;
}

void tell_me(int f(const char *, const char *))
{
   if (f == strcmp)
      printf("Address of strcmp(): %p\n", f);
   else
      printf("Function address: %p\n", f);
}
```

Because the pointer being passed in the second call to `tell_me()` has the wrong type, your compiler should complain. Does it? Modify the program by changing the pointer type throughout to the generic `void*` type. Does this make your compiler happy?

43 (Advanced) The following program has an error in it:

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

int main(void)
{
   char   *p1 = "abc", *p2 = "pacific sea";

   printf("%s   %s   %s\n", p1, p2, strcat(p1, p2));
   return 0;
}
```

On our system, what happens is compiler-dependent. With one of our compilers, the program exhibits a run-time error. With another compiler, we get the following written to the screen:

```
abcpacific sea   acific sea   abcpacific sea
```

This output makes sense and tells us something about the compiler. What programming error did we make? What does the output tell us about our compiler? Which is the preferred behavior: a compiler that produces executable code that exhibits a run-time error, or a compiler that produces (sometimes) logically incorrect output?

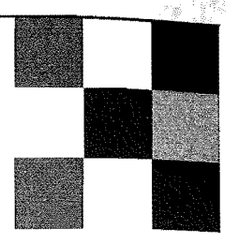
44 A function name by itself is treated by the compiler as a pointer. This is a general rule in C. Is the following code legal?

```
#include <stdio.h>

void   f(void);
void   g(void);
void   h(void);

int main(void)
{
   (*f)();
   return 0;
}

void f(void)
{
   printf("Hello from f().\n");
   (((*g)))();
}

void g(void)
{
   printf("Hello from g().\n");
   (*(*(*h)))();
}

void h(void)
{
   printf("Hello from h().\n");
}
```

Write your answer before you try to compile the program. Your answer should be based on your general knowledge about how well a C compiler can be expected to follow a rule, even if the rule is applied in some strange way.

45 A precondition for the `root()` solver to work is that `f(a)` and `f(b)` must have opposite signs. Use an assertion at the beginning of `root()` to check this condition.

46 The ancient Egyptians wrote in hieroglyphics. In this system of writing, vowel sounds are not represented, only consonants. Is written English generally understandable without vowels? To experiment, write a function `is_vowel()` that returns 1 if a letter is a vowel and 0 otherwise. Use your function in a program that reads the standard input file and writes to the standard output file, deleting all vowels. Use redirection on a file containing some English text to test your program.



Chapter 7

Bitwise Operators and Enumeration Types

There are two additional ways to represent discrete values: as bits and as elements in a finite set. In this chapter, we first discuss the bitwise operators. Even though expressions involving bitwise operators are explicitly system-dependent, they are very useful. We illustrate their usefulness in packing and unpacking data.

In the second half of the chapter, we discuss the enumeration types. Enumeration types are user-defined types that allow the programmer to name a finite set together with its elements, which are called *enumerators*. These types are defined and used by the programmer as the need arises. We illustrate much of this material by implementing a completely worked out interactive game program.

7.1 Bitwise Operators and Expressions

The bitwise operators act on integral expressions represented as strings of binary digits. These operators are explicitly system-dependent. We will restrict our discussion to machines having 8-bit bytes, 4-byte words, the two's complement representation of integers, and ASCII character codes.

Bitwise operators		
Logical operators	(unary) bitwise complement	~
	bitwise and	&
	bitwise exclusive or	^
	bitwise inclusive or	|
Shift operators	left shift	<<
	right shift	>>

Like other operators, the bitwise operators have rules of precedence and associativity that determine how expressions involving them are evaluated.

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

The operator ~ is unary; all the other bitwise operators are binary. They operate on integral expressions. We will discuss each of the bitwise operators in detail.

Bitwise Complement

The operator ~ is called the *one's complement* operator, or the *bitwise complement* operator. It inverts the bit string representation of its argument; the 0s become 1s, and the 1s become 0s. Consider, for example, the declaration

```
int   a = 70707;
```

The binary representation of a is

```
00000000 00000001 00010100 00110011
```

The expression ~a is the bitwise complement of a, and this expression has the binary representation

```
11111111 11111110 11101011 11001100
```

The int value of the expression ~a is -70708.

Two's Complement

The *two's complement representation* of a nonnegative integer *n* is the bit string obtained by writing *n* in base 2. If we take the bitwise complement of the bit string and add 1 to it, we obtain the two's complement representation of *-n*. The next table gives some examples. To save space, we show only the two low-order bytes.

Value of n	Binary representation	Bitwise complement	Two's complement representation of -n	Value of -n
7	00000000 00000111	11111111 11111000	11111111 11111001	-7
8	00000000 00001000	11111111 11110111	11111111 11111000	-8
9	00000000 00001001	11111111 11110110	11111111 11110111	-9
-7	11111111 11111001	00000000 00000110	00000000 00000111	7

The preceding table is read from left to right. If we start with a positive integer *n*, consider its binary representation, and take its bitwise complement and add 1, then we

obtain the two's complement representation of *-n*. A machine that uses the two's complement representation as its binary representation in memory for integral values is called a *two's complement machine.*

On a two's complement machine, if we start with the binary representation of a negative number *-n* and take its bitwise complement and add 1, we obtain the two's complement representation, or binary representation, of *n*. This is illustrated in the last line in the preceding table.

The two's complement representations of both 0 and -1 are special. The value 0 has all bits off; the value -1 has all bits on. Note that if a binary string is added to its bitwise complement, then the result has all bits on, which is the two's complement representation of -1. Negative numbers are characterized by having the high bit on.

On a two's complement machine, the hardware that does addition and bitwise complementation can be used to implement subtraction. The operation `a - b` is the same as `a + (-b)`, and `-b` is obtained by taking the bitwise complement of `b` and adding 1.

Bitwise Binary Logical Operators

The three operators & (and), ^ (exclusive or), and | (inclusive or) are binary operators. They take integral expressions as operands. The two operands, properly widened, are operated on bit position by bit position. The following table shows the bitwise operators acting on 1-bit fields. The table defines the semantics of the operators.

Values of:				
a	b	a & b	a ^ b	a | b
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

The next table contains examples of the bitwise operators acting on `int` variables.

Declaration and initializations		
`int a = 33333, b = -77777;`		
Expression	Representation	Value
a	`00000000 00000000 10000010 00110101`	33333
b	`11111111 11111110 11010000 00101111`	-77777
a & b	`00000000 00000000 10000000 00100101`	32805
a ^ b	`11111111 11111110 01010010 00011010`	-110054
a | b	`11111111 11111110 11010010 00111111`	-77249
~(a | b)	`00000000 00000001 00101101 11000000`	77248
(~a & ~b)	`00000000 00000001 00101101 11000000`	77248

Left and Right Shift Operators

The two operands of a shift operator must be integral expressions. The integral promotions are performed on each of the operands. The type of the expression as a whole is that of its promoted left operand. An expression of the form

expr1 << *expr2*

causes the bit representation of *expr1* to be shifted to the left by the number of places specified by *expr2*. On the low-order end, 0s are shifted in.

Declaration and initialization		
`char c = 'Z';`		
Expression	Representation	Action
c	`00000000 00000000 00000000 01011010`	unshifted
c << 1	`00000000 00000000 00000000 10110100`	left-shifted 1
c << 4	`00000000 00000000 00000101 10100000`	left-shifted 4
c << 31	`00000000 00000000 00000000 00000000`	left-shifted 31

Even though `c` is stored in 1 byte, in an expression it gets promoted to an `int`. When shift expressions are evaluated, integral promotions are performed on the two oper-

ands separately, and the type of the result is that of the promoted left operand. Thus, the value of an expression such as `c << 1` gets stored in 4 bytes.

The right shift operator >> is not symmetric to the left shift operator. For unsigned integral expressions, 0s are shifted in at the high end. For the signed types, some machines shift in 0s, while others shift in sign bits. (See exercise 4, on page 357.) The sign bit is the high-order bit; it is 0 for nonnegative integers and 1 for negative integers.

Declarations and initializations		
`int` `unsigned`	`a = 1 << 31;  /* shift 1 to the high bit */` `b = 1 << 31;`	
Expression	Representation	Action
a	10000000 00000000 00000000 00000000	unshifted
a >> 3	11110000 00000000 00000000 00000000	right-shifted 3
b	10000000 00000000 00000000 00000000	unshifted
b >> 3	00010000 00000000 00000000 00000000	right-shifted 3

Note that on our machine, sign bits are shifted in with an `int`. On another machine, 0s might be shifted in. To avoid this difficulty, programmers often use unsigned types when using bitwise operators.

If the right operand of a shift operator is negative or has a value that equals or exceeds the number of bits used to represent the left operand, then the behavior is undefined. It is the programmer's responsibility to keep the value of the right operand within proper bounds.

Our next table illustrates the rules of precedence and associativity with respect to the shift operators. To save space, we show only the two low-order bytes.

Declaration and assignments			
`unsigned  a = 1, b = 2;`			
Expression	Equivalent expression	Representation	Value
a << b >> 1	(a << b) >> 1	00000000 00000010	2
a << 1 + 2 << 3	(a << (1 + 2)) << 3	00000000 01000000	64
a + b << 12 * a >> b	((a + b) << (12 * a)) >> b	00001100 00000000	3072

In C++, the two shift operators are *overloaded* and used for input/output. Overloading in C++ is a method of giving existing operators and functions additional meanings. (See Section 13.5, "Overloading," on page 603, for examples and explanation.)

7.2 Masks

A mask is a constant or variable that is used to extract desired bits from another variable or expression. Because the `int` constant 1 has the bit representation

```
00000000 00000000 00000000 00000001
```

it can be used to determine the low-order bit of an `int` expression. The following code uses this mask and prints an alternating sequence of 0s and 1s:

```
int   i, mask = 1;
for (i = 0; i < 10; ++i)
   printf("%d", i & mask);
```

If we wish to find the value of a particular bit in an expression, we can use a mask that is 1 in that position and 0 elsewhere. For example, we can use the expression `1 << 2`, as a mask for the third bit, counting from the right. The expression

```
(v & (1 << 2)) ? 1 : 0
```

has the value 1 or 0 depending on the third bit in `v`.

Another example of a mask is the constant value 255, which is $2^8 - 1$. It has the following bit representation:

```
00000000 00000000 00000000 11111111
```

Because only the low-order byte is turned on, the expression

```
v & 255
```

will yield a value having a bit representation with all its high-order bytes zero and its low-order byte the same as the low-order byte in `v`. We express this by saying, "255 is a mask for the low-order byte."

7.3 Software Tools: Printing an int Bitwise

Software tools are utilities that the programmer can use to write software. Most systems provide a variety of software tools. Examples are compilers, debuggers, and the *make* utility. We will discuss these in Chapter 11, "Input/Output and the Operating System." Programmers often write other software tools for their own use as the need arises. The `bit_print()` function that we discuss in this section is a typical example. For anyone writing software that deals with the machine at the bit level, the `bit_print()` utility is essential; it allows the programmer to see what is happening. For the beginning programmer, exploration with `bit_print()` helps to provide a conceptual framework that is very useful.

Our `bit_print()` function uses a mask to print out the bit representation of an `int`. The function can be used to explore how values of expressions are represented in memory. We used it, in fact, to help create the tables in this chapter.

In file bit_print.c

```
/* Bit print an int expression. */

#include <limits.h>

void bit_print(int a)
{
   int   i;
   int   n = sizeof(int) * CHAR_BIT;        /* in limits.h */
   int   mask = 1 << (n - 1);               /* mask = 100...0 */

   for (i = 1; i <= n; ++i) {
      putchar(((a & mask) == 0) ? '0' : '1');
      a <<= 1;
      if (i % CHAR_BIT == 0 && i < n)
       putchar(' ');
   }
}
```


Dissection of the bit_print() Function

```
#include <limits.h>
```

In ANSI C, the symbolic constant CHAR_BIT is defined in *limits.h*. In traditional C, this header file is not usually available. The value of CHAR_BIT on most systems is 8. It represents the number of bits in a char, or equivalently, the number of bits in a byte. ANSI C requires at least 8 bits in a byte.

```
int   n = sizeof(int) * CHAR_BIT;        /* in limits.h */
```

Because we want this function to work on machines having either 2- or 4-byte words, we use the variable `n` to represent the number of bits in a machine word. We expect the value of the expression `sizeof(int)` to be either 2 or 4, and we expect that the symbolic constant CHAR_BIT, which is defined in the standard header file *limits.h*, will be 8. Thus, we expect `n` to be initialized to either 16 or 32, depending on the machine.

```
int   mask = 1 << (n - 1);               /* mask = 100...0 */
```

Because of operator precedence, the parentheses are not needed in the initialization. We put them there to make the code more readable. Because << has higher precedence than = , the expression `1 << (n - 1))` gets evaluated first. Suppose that `n` has value 32. The constant `1` has only its low-order bit turned on. The expression `1 << 31` shifts that bit to the high-order end. Thus, `mask` has all its bits off except for its high-order bit (sign bit), which is on.

```
for (i = 1; i <= n; ++i) {
   putchar(((a & mask) == 0) ? '0' : '1');
   a <<= 1;
   .....
```

First consider the expression

```
(a & mask) == 0
```

If the high-order bit in a is off, then the expression a & mask has all its bits off, and the expression (a & mask) == 0 is *true*. Conversely, if the high-order bit in a is on, then the expression a & mask has its high-order bit on, and the expression (a & mask) == 0 is *false*. Now consider the expression

```
((a & mask) == 0) ? '0' : '1'
```

If the high-order bit in a is off, then the conditional expression has the value '0'; otherwise, it has the value '1'. Thus, putchar() prints a 0 if the high-order bit is off and prints a 1 if it is on.

```
putchar(((a & mask) == 0) ? '0' : '1');
a <<= 1;
```

After the high-order bit in a has been printed, we left-shift the bits in a by 1 and place the result back in a. Recall that

```
a <<= 1;    is equivalent to    a = a << 1;
```

The value of the expression a << 1 has the same bit pattern as a, except that it has been left-shifted by 1. The expression by itself does not change the value of a in memory. In contrast to this, the expression a <<= 1 does change the value of a in memory. Its effect is to bring the next bit into the high-order position, ready to be printed the next time through the loop.

```
if (i % CHAR_BIT == 0 && i < n)
   putchar(' ');
```

If we assume that the value of the symbolic constant CHAR_BIT is 8, then this code causes a blank to be printed after each group of 8 bits has been printed. It is not necessary to do this, but it certainly makes the output easier to read.

7.4 Packing and Unpacking

The use of bitwise expressions allows for data compression across byte boundaries. This is useful in saving space, but it can be even more useful in saving time. On a machine with 4-byte words, each instruction cycle processes 32 bits in parallel. The following function can be used to pack four characters into an int. It uses shift operations to do the packing byte by byte.

In file pack_bits.c

```
/* Pack 4 characters into an int. */

#include <limits.h>

int pack(char a, char b, char c, char d)
{
   int   p = a;        /* p will be packed with a, b, c, d */

   p = (p << CHAR_BIT) | b;
   p = (p << CHAR_BIT) | c;
   p = (p << CHAR_BIT) | d;
   return p;
}
```

To test our function, we write a program with the lines

```
printf("abcd = ");
bit_print(pack('a', 'b', 'c', 'd'));
putchar('\n');
```

in main(). Here is the output of our test program:

```
abcd = 01100001 01100010 01100011 01100100
```

Observe that the high-order byte has value 97, or 'a', and that the values of the remaining bytes are 98, 99, and 100. Thus, pack() did its work properly.

Having written pack(), we now want to be able to retrieve the characters from within the 32-bit int. Again, we can use a mask to do this.

```
/* Unpack a byte from an int. */

#include <limits.h>

char unpack(int p, int k)              /* k = 0, 1, 2, or 3 */
{
   int        n = k * CHAR_BIT;        /* n = 0, 8, 16, or 24 */
   unsigned   mask = 255;              /* low-order byte */

   mask <<= n;
   return ((p & mask) >> n);
}
```

Dissection of the unpack() Function

■ `#include <limits.h>`

We have included this header file because it contains the definition of the symbolic constant CHAR_BIT. It represents the number of bits in a byte. On most machines its value is 8.

■
```
char unpack(int p, int k)              /* k = 0, 1, 2, or 3 */
{
   .....
```

We think of the parameter p as a packed int with its bytes numbered 0 through 3. The parameter k will indicate which byte we want: If k has value 0, then we want the low-order byte; if k has value 1, then we want the next byte; and so forth.

■ `int        n = k * CHAR_BIT;        /* n = 0, 8, 16, or 24 */`

If we assume that CHAR_BIT is 8 and that k has value 0, 1, 2, or 3, then n will be initialized with the value 0, 8, 16, or 24.

■ `unsigned   mask = 255;              /* low-order byte */`

The constant 255 is special; to understand it, first consider 256. Because $256 = 2^8$, the bit representation of 256 has all bits 0 except for a 1 in the 9th bit, counting from the low-order bit. Because 255 is one less than 256, the bit representation of 255 has all bits 0, except for the first 8 bits, which are all 1. (See exercise 8, on page 358.) Thus, the binary representation of mask is


```
00000000 00000000 00000000 11111111
```

■ `mask <<= n;`

Let us assume that CHAR_BIT is 8. If n has value 0, then the bits in mask are not changed. If n has value 8, then the bits in mask are left-shifted by 8. In this case we think of mask stored in memory as

```
00000000 00000000 11111111 00000000
```

If n has value 16, then the bits in mask are left-shifted by 16. In this case we think of mask stored in memory as

```
00000000 11111111 00000000 00000000
```

In a similar fashion, if n has value 24, then mask will have only the bits in its high-order byte turned on.

■ `(p & mask) >> n`

Parentheses are needed because & has lower precedence than >>. Suppose that p has value -3579753 (which we chose because it has a suitable bit pattern), and suppose that n has value 16. The following table illustrates what happens:

Expression	Binary representation	Value
p	11111111 11001001 01100000 10010111	-3579753
mask	00000000 11111111 00000000 00000000	16711680
p & mask	00000000 11001001 00000000 00000000	13172736
(p & mask) >> n	00000000 00000000 00000000 11001001	201

■ `return ((p & mask) >> n);`

Because the function type for unpack() is char, the int expression (p & mask) >> n gets converted to a char before it gets passed back to the calling environment. When an int is converted to a char, only the low-order byte is kept; other bytes are discarded.

Imagine wanting to keep an abbreviated employee record in one integer. We will suppose that an "employee identification number" can be stored in 9 bits and that a "job type" can be stored in 6 bits, which provides for a total of up to 64 different job types.

The employee's "gender" can be stored in 1 bit. These three fields will require 16 bits, which, on a machine with 4-byte words, is a `short` integer. We can think of the three bit fields as follows:

Identification	Job type	Gender
bbbbbbbbb	bbbbbb	b

The following function can be used in a program designed to enter employee data into a `short`. The inverse problem of reading data out of the `short` would be accomplished with the use of masks.

```
/* Create employee data in a short int. */

short create_employee_data(int id_no, int job_type, char gender)
{
   short   employee = 0;     /* start with all bits off */

   employee |= (gender == 'm' || gender == 'M') ? 0 : 1;
   employee |= job_type << 1;
   employee |= id_no << 7;
   return employee;
}
```

Multibyte Character Constants

Multibyte characters are allowed in ANSI C. An example is `'abc'`. On a machine with 4-byte words, this causes the characters `'a'`, `'b'`, and `'c'` to be packed into a single word. However, the order in which they are packed is machine-dependent. Some machines put `'a'` in the low-order byte; others put it in the high-order byte. (See exercise 12, on page 359.)

7.5 Enumeration Types

The keyword `enum` is used to declare enumeration types. It provides a means of naming a finite set, and of declaring identifiers as elements of the set. Consider, for example, the declaration

```
enum day {sun, mon, tue, wed, thu, fri, sat};
```

This creates the user-defined type `enum day`. The keyword `enum` is followed by the tag name `day`. The enumerators are the identifiers `sun`, `mon`, ... , `sat`. They are constants of type `int`. By default, the first one is 0, and each succeeding one has the next integer value. This declaration is an example of a type specifier, which we also think of as a *template*. No variables of type `enum day` have been declared yet. To do so, we can now write

```
enum day   d1, d2;
```

This declares `d1` and `d2` to be of type `enum day`. They can take on as values only the elements (enumerators) in the set. Thus,

```
d1 = fri;
```

assigns the value `fri` to `d1`, and

```
if (d1 == d2)
   .....        /* do something */
```

tests whether `d1` is equal to `d2`. Note carefully that the type is `enum day`. The keyword `enum` by itself is not a type.

The enumerators can be initialized. Also, we can declare variables along with the template, if we wish to do so. The following is an example:

```
enum suit {clubs = 1, diamonds, hearts, spades}   a, b, c;
```

Because `clubs` has been initialized to 1, `diamonds`, `hearts`, and `spades` have the values 2, 3, and 4, respectively. In this example

```
enum suit {clubs = 1, diamonds, hearts, spades}
```

is the type specifier, and a, b, and c are variables of this type. Here is another example of initialization:

```
enum fruit {apple = 7, pear, orange = 3, lemon}   frt;
```

Because the enumerator apple has been initialized to 7, pear has value 8. Similarly, because orange has value 3, lemon has value 4. Multiple values are allowed, but the identifiers themselves must be unique.

```
enum veg {beet = 17, carrot = 17, corn = 17}   vege1, vege2;
```

The tag name need not be present. Consider, for example,

```
enum {fir, pine}   tree;
```

Because there is no tag name, no other variables of type enum {fir, pine} can be declared.

The following is the syntax for the enumeration declaration:

enum_declaration ::= *enum_type_specifier* *identifier* { , *identifier* }$_{0+}$;
enum_type_specifier ::= enum *e_tag* { *e_list* }
| enum *e_tag*
| enum { *e_list* }
e_tag ::= *identifier*
e_list ::= *enumerator* { , *enumerator* }$_{0+}$
enumerator ::= *identifier* { = *constant_integral_expression* }$_{opt}$

In general, one should treat enumerators as programmer-specified constants and use them to aid program clarity. If necessary, the underlying value of an enumerator can be obtained by using a cast. The variables and enumerators in a function must all have distinct identifiers. The tag names, however, have their own name space. This means that we can reuse a tag name as a variable or as an enumerator. The following is an example:

```
enum veg {beet, carrot, corn}   veg;
```

Although this is legal, it is not considered good programming practice.

We illustrate the use of the enumeration type by writing a function that computes the next day which uses typedef to replace the enum keyword in the type declaration.

In file next_day.c

```
/* Compute the next day. */

enum day {sun, mon, tue, wed, thu, fri, sat};

typedef   enum day   day;   /*the usual typedef trick */

day find_next_day(day d)
{
   day   next_day;

   switch (d) {
   case sun:
      next_day = mon;
      break;
   case mon:
      next_day = tue;
      break;
   case tue:
      next_day = wed;
      break;
   case wed:
      next_day = thu;
      break;
   case thu:
      next_day = fri;
      break;
   case fri:
      next_day = sat;
      break;
   case sat:
      next_day = sun;
      break;
   }
   return next_day;
}
```

Recall that only a constant integral expression can be used in a case label. Because enumerators are constants, they can be used in this context. The following is another version of this function; this version uses a cast to accomplish the same ends:

```
/* Compute the next day with a cast. */

enum day {sun, mon, tue, wed, thu, fri, sat};

typedef   enum day   day;

day find_next_day(day d)
{
   assert((int) d >= 0 && (int) d < 7)
   return ((day)(((int) d + 1) % 7));
}
```

Enumeration types can be used in ordinary expressions provided type compatibility is maintained. However, if one uses them as a form of integer type and constantly accesses their implicit representation, it is better just to use integer variables instead. The importance of enumeration types is their self-documenting character, where the enumerators are themselves mnemonic. Furthermore, enumerators force the compiler to provide programmer-defined type checking so that one does not inadvertently mix apples and diamonds.

7.6 An Example: The Game of Paper, Rock, Scissors

We will illustrate some of the concepts introduced in this chapter by writing a program to play the traditional children's game called "paper, rock, scissors." In this game each child uses her or his hand to represent one of the three objects. A flat hand held in a horizontal position represents "paper," a fist represents "rock," and two extended fingers represent "scissors." The children face each other and at the count of three display their choices. If the choices are the same, then the game is a tie. Otherwise, a win is determined by the rules:

Paper, Rock, Scissors Rules

- Paper covers the rock.
- Rock breaks the scissors.
- Scissors cut the paper.

We will write this program in its own directory. The program will consist of a *.h* file and a number of *.c* files. Each of the *.c* files will include the header file at the top of the

file. In the header file we put `#include` directives, templates for our enumeration types, type definitions, and function prototypes:

In file p_r_s.h

```
/* The game of paper, rock, scissors. */

#include <ctype.h>        /* for isspace() */
#include <stdio.h>        /* for printf(), etc */
#include <stdlib.h>       /* for rand() and srand() */
#include <time.h>         /* for time() */

enum p_r_s {paper, rock, scissors,
          game, help, instructions, quit};
enum outcome {win, lose, tie, error};

typedef   enum p_r_s     p_r_s;
typedef   enum outcome   outcome;

outcome   compare(p_r_s player_choice,
        p_r_s machine_choice);
void      prn_final_status(int win_cnt, int lose_cnt);
void      prn_game_status(int win_cnt,
              int lose_cnt, int tie_cnt);
void      prn_help(void);
void      prn_instructions(void);
void      report_and_tabulate(outcome result,
                  int *win_cnt_ptr,
                  int *lose_cnt_ptr,
                  int *tie_cnt_ptr);
p_r_s     selection_by_machine(void);
p_r_s     selection_by_player(void);
```

We do not normally comment our `#include` lines, but here we are trying to make the code more readable for the novice programmer. Here is our `main()` function:

In file main.c

```
#include "p_r_s.h"

int main(void)
{
   int       win_cnt = 0, lose_cnt = 0, tie_cnt = 0;
   outcome   result;
   p_r_s     player_choice, machine_choice;

   srand(time(NULL));     /* seed the random number generator */
   prn_instructions();
   while ((player_choice = selection_by_player()) != quit)
      switch (player_choice) {
      case paper:
      case rock:
      case scissors:
         machine_choice = selection_by_machine();
       result = compare(player_choice, machine_choice);
       report_and_tabulate(result, &win_cnt, &lose_cnt,
                           &tie_cnt);
         break;
      case game:
         prn_game_status(win_cnt, lose_cnt, tie_cnt);
         break;
      case instructions:
         prn_instructions();
         break;
      case help:
         prn_help();
         break;
      default:
       printf("\nPROGRAMMER ERROR: Cannot get to here!\n\n");
       exit(1);
      }
   prn_game_status(win_cnt, lose_cnt, tie_cnt);
   prn_final_status(win_cnt, lose_cnt);
   return 0;
}
```

The first executable statement in `main()` is

```
srand(time(NULL));
```

This seeds the random-number generator `rand()`, causing it to produce a different sequence of integers each time the program is executed. More explicitly, passing `srand()` an integer value determines where `rand()` will start. The function call `time(NULL)` returns a count of the number of seconds that have elapsed since 1 January 1970 (the approximate birthday of UNIX). Both `srand()` and `time()` are provided in the standard library. The function prototype for `srand()` is in *stdlib.h*, and the function prototype for `time()` is in *time.h*. Both of these header files are provided by the system. Note that we included them in *p_r_s.h*.

The next executable statement in `main()` calls `prn_instructions()`. This provides instructions to the user. Embedded in the instructions are some of the design considerations for programming this game. We wrote this function, along with other printing functions, in *prn.c*:

In file prn.c

```
#include "p_r_s.h"

void prn_final_status(int win_cnt, int lose_cnt)
{
   if (win_cnt > lose_cnt)
      printf("CONGRATULATIONS - You won!\n\n");
   else if (win_cnt == lose_cnt)
      printf("A DRAW - You tied!\n\n");
   else
      printf("SORRY - You lost!\n\n");
}

void prn_game_status(int win_cnt, int lose_cnt, int tie_cnt)
{
   printf("\n%s\n%s%4d\n%s%4d\n%s%4d\n%s%4d\n\n",
      "GAME STATUS:",
      "   Win:   ", win_cnt,
      "   Lose:  ", lose_cnt,
      "   Tie:   ", tie_cnt,
      "   Total: ", win_cnt + lose_cnt + tie_cnt);
}

void prn_help(void)
{
   printf("\n%s\n",
      "The following characters can be used for input:\n"
      "      p   for paper\n"
      "      r   for rock\n"
      "      s   for scissors\n"
      "      g   print the game status\n"
      "      h   help, print this list\n"
      "      i   reprint the instructions\n"
      "      q   quit this game\n");
}
```

```
void prn_instructions(void)
{
   printf("\n%s\n",
      "PAPER, ROCK, SCISSORS:\n"
      "   In this game p is for \"paper,\" r is for \"rock,\" and"
      " s is for \"scissors.\"\n"
      "   Both the player and the machine\n"
      "   will choose one of p, r, or s."
      "  If the two choices are the same,\n"
      "   then the game is a tie.  Otherwise:\n"
      "      \"paper covers the rock\"     (a win for paper),\n"
      "      \"rock breaks the scissors\"  (a win for rock),\n"
      "      \"scissors cut the paper\"    (a win for scissors).\n"
      "\n"
      "   There are other allowable inputs:\n"
      "      g   for game status   (the number of wins so far),\n"
      "      h   for help,\n"
      "      i   for instructions  (reprint these instructions),\n"
      "      q   for quit          (to quit the game).\n"
      "\n"
      "   This game is played repeatedly until q is entered.\n"
      "\n"
      "   Good luck!\n");
}
```

To play the game, both the machine and the player (user) need to make a selection from "paper, rock, scissors." We write these routines in *selection.c*:

In file selection.c

```
#include "p_r_s.h"

p_r_s selection_by_machine(void)
{
   return ((p_r_s) (rand() % 3));
}
```



```
p_r_s selection_by_player(void)
{
   char    c;
   p_r_s   player_choice;

   printf("Input p, r, or s:  ");
   while (isspace(c = getchar()))     /* skip white space */
      ;
   switch (c) {
   case 'p':
      player_choice = paper;
      break;
   case 'r':
      player_choice = rock;
      break;
   case 's':
      player_choice = scissors;
      break;
   case 'g':
      player_choice = game;
      break;
   case 'i':
      player_choice = instructions;
      break;
   case 'q':
      player_choice = quit;
      break;
   default:
      player_choice = help;
      break;
   }
   return player_choice;
}
```

The machine's selection is computed by the uses of the expression `rand() % 3` to produce a randomly distributed integer between 0 and 2. Because the type of the function is `p_s_r`, the value returned will be converted to this type, if necessary. We provided an explicit cast to make the code more self-documenting.

Note that in `selection_by_player()` we use the macro `isspace()` from *ctype.h* to skip white space. (See Section 8.7, "The Macros in stdio.h and ctype.h," on page 382.) After white space is skipped, all other characters input at the terminal are processed, most of them through the `default` case of the `switch` statement.

The value returned by `selection_by_player()` determines which case gets executed in the `switch` statement in `main()`. The value returned depends on what the player types. If the character `g` is input, then `prn_game_status()` is invoked; if any character other than white space or `p`, `r`, `s`, `g`, `i`, or `q` is input, then `prn_help()` is invoked.

Once the player and the machine have made a selection, we need to compare the two selections in order to determine the outcome of the game. The following function does this:

In file compare.c

```
#include "p_r_s.h"

outcome compare(p_r_s player_choice, p_r_s machine_choice)
{
   outcome   result;

   if (player_choice == machine_choice)
      return tie;
   switch (player_choice) {
   case paper:
      result = (machine_choice == rock) ? win : lose;
      break;
   case rock:
      result = (machine_choice == scissors) ? win : lose;
      break;
   case scissors:
      result = (machine_choice == paper) ? win : lose;
      break;
   default:
      printf("\nPROGRAMMER ERROR: Unexpected choice!\n\n");
      exit(1);
   }
   return result;
}
```

The value returned by the call to `compare()` in `main()` gets passed to the function `report_and_tabulate()`. This function reports to the user the result of a round of play and increments as appropriate the number of wins, losses, and ties.

In file report.c

```
#include "p_r_s.h"

void report_and_tabulate(outcome result,
     int *win_cnt_ptr, int *lose_cnt_ptr, int *tie_cnt_ptr)
{
   switch (result) {
   case win:
      ++*win_cnt_ptr;
      printf("%27sYou win.\n", "");
      break;
   case lose:
      ++*lose_cnt_ptr;
      printf("%27sYou lose.\n", "");
      break;
   case tie:
      ++*tie_cnt_ptr;
      printf("%27sA tie.\n", "");
      break;
   default:
      printf("\nPROGRAMMER ERROR: Unexpected result!\n\n");
      exit(1);
   }
}
```

We are now ready to compile our function. We can do this with the command

```
cc -o p_r_s main.c compare.c prn.c report.c selection.c
```

Later, after we have learned about the *make* utility, we can facilitate program development by using an appropriate makefile. (See Section 11.17, "The Use of make," on page 532.)

Summary

1 The bitwise operators provide the programmer with a means of accessing the bits in an integral expression. Typically, we think of the operands of these operators as bit strings.

2 The use of bitwise expressions allows for data compression across byte boundaries. This capability is useful in saving space, but is more useful in saving time. On a machine with 4-byte words, each instruction cycle processes 32 bits in parallel.

3 Most machines use the two's complement representation for integers. In this representation, the high-order bit is the sign bit. It is 1 for negative integers and 0 for nonnegative integers.

4 Bitwise operations are explicitly machine-dependent. A left shift causes 0s to be shifted in. The situation for a right shift is more complicated. If the integral expression is `unsigned`, then 0s are shifted in. If the expression is one of the signed types, then what gets shifted in is machine-dependent. Some machines shift in sign bits. This means that if the sign bit is 0, then 0s are shifted in, and if the sign bit is 1, then 1s are shifted in. Some machines shift in 0s in all cases.

5 Masks are particular values used typically with the & operator to extract a given series of bits. Packing is the act of placing a number of distinct values into various subfields of a given variable. Unpacking extracts these values.

6 The keyword `enum` allows the programmer to define enumeration types. A variable of such a type can take values from the set of enumerators associated with the type.

7 Enumerators are distinct identifiers chosen for their mnemonic significance. Their use provides a type-checking constraint for the programmer, as well as self-documentation for the program.

8 Enumerators are constants of type `int`. Thus, they can be used in `case` labels in a `switch`. A cast can be used to resolve type conflicts.

Exercises

1 Suppose that integers have a 16-bit two's complement representation. Write the binary representation for -1, -5, -101, -1023. Recall that the two's complement representation of negative integers is obtained by taking the bit representation of the corresponding positive integer, complementing it, and adding 1.

2 Alice, Betty, and Carole all vote on 16 separate referenda. Assume that each individual's vote is stored bitwise in a 16-bit integer. Write a function definition that begins

```
short majority(short a, short b, short c)
{
   .....
```

This function should take as input the votes of Alice, Betty, and Carole stored in `a`, `b`, and `c`, respectively. It should return the bitwise majority of `a`, `b`, and `c`.

3 Write a function definition that begins

```
int circular_shift(int a, int n)
{
   .....
```

This function should left-shift `a` by `n` positions, where the high-order bits are reintroduced as the low-order bits. Here are two examples of a circular shift operation defined for a `char` instead of an `int`:

```
10000001    circular shift 1 yields    00000011
01101011    circular shift 3 yields    01011011
```

4 Does your machine shift in sign bits? Here is some code that will help you determine this. Explain why this code works.

```
int        i = -1;        /* turn all bits on */
unsigned   u = -1;

if (i >> 1 == u >> 1)
   printf("Zeros are shifted in.\n");
else
   printf("Sign bits are shifted in.\n");
```

5 Write a function that will reverse the bit representation of an `int`. Here are two examples of a reversing operation defined for a `char` instead of an `int`:

```
01110101    reversed yields    10101110
10101111    reversed yields    11110101
```

6 Write a function that will extract every other bit position from a 32-bit expression. The result should be returned as a 16-bit expression. Your function should work on machines having either 2- or 4-byte words.

7 Write a function that takes as its input a string of decimal integers. Each character in the string can be thought of as a decimal digit. The digits should be converted to 4-bit binary strings and packed into an `int`. If an `int` has 32 bits, then eight digits can be packed into it. When you test your function, here is what you might see on the screen:

```
Input a string of decimal digits:  12345678
12345678 = 0001 0010 0011 0100 0101 0110 0111 1000
```

Also, write an inverse function. It should unpack an `int` and return the original string. *Hint:* Here is one way to begin a conversion function:

```
int convert(char *s)
{
   char   *p;
   int    a = 0;   /* turn all bits off */

   for (p = s; *p != '\0'; ++p) {
      a <<= 4;
      switch (*p) {
      case '1':
         a |= 1;
       break;
      case '2':
      .....
```

8 Use the `bit_print()` function to create a table containing n, the binary representation for $2n$, and the binary representation for 2^n, for $n = 0, 1, 2, \ldots, 32$. If your machine has 2-byte words, then the output of your program should look like this:

```
 0:  00000000 00000001   00000000 00000000
 1:  00000000 00000010   00000000 00000001
 2:  00000000 00000100   00000000 00000011
 .....
15:  10000000 00000000   01111111 11111111
.....
```


After you have done this, write down a similar table by hand that contains n, 10^n, and $10^n - 1$ for $n = 0, 1, 2, \ldots, 7$. Write the numbers in base 10 in your table. Do you see the similarity between the two tables? *Hint:* Use the following code:

```
int   i, power = 1;

for (i = 0; i < 32; ++i) {
   printf("%2d:  ", i);
   bit_print(power);
   printf("   ");
   bit_print(power - 1);
   putchar('\n');
   power *= 2;
}
```

9 Some of the binary representations in the tables in this chapter are easy to check for correctness, and some are not. Use `bit_print()` to check some of the more difficult representations.

10 Write a version of the `bit_print()` function that will work on machines with either 2- or 4-byte words. *Hint:* Use the `sizeof` operator to find the number of bytes in an `int`.

11 If you are not familiar with the use of the constants `0xff`, `0xff00`, `0xff0000`, and `0xff000000` as masks, write a test program that uses `bit_print()` to print these values as bit strings.

12 If your machine has 4-byte words, use the function `bit_print()` to find out how the multibyte character `'abc'` is stored on your machine. If your machine has 2-byte words, then you can put only two characters into a multibyte character. In that case, try `'ab'`.

13 Write a roulette program. The roulette (machine) will select a number between 0 and 35 at random. The player can place an odd/even bet, or can place a bet on a particular number. A winning odd/even bet is paid off at 2 to 1, except that all odd/even bets lose if the roulette selects 0. If the player places a bet on a particular number and the roulette selects it, then the player is paid off at 35 to 1. If you play this game, how many one-dollar bets can you place before you lose ten dollars?

14 Write a function called `previous_month()` that returns the previous month. Start with the code

```
enum month {jan, feb, ... , dec};

typedef   enum month   month;
```

If `jan` is passed as an argument to the function, then `dec` should be returned. Write another function that prints the name of a month. More explicitly, if the enumerator `jan` is passed as an argument, then `January` should be printed. Write `main()` so that it calls your functions and produces a table of all twelve months, each one listed next to its predecessor month. *Caution:* When `printf()` is used, a variable of an enumeration type is printed as its implicit integer value. That is,

```
printf("%d\n", jan);
```

prints `0`, not `jan`.

15 Write a next-day program for a particular year. The program should take as input two integers, say 17 and 5, which represents 17 May, and it should print as output `18 May`, which is the next day. Use enumeration types in the program. Pay particular attention to the problem of crossing from one month to the next.

16 A twentieth-century date can be written with integers in the form *day/month/year*. An example is 1/7/33, which represents 1 July 1933. Write a function that stores the day, month, and year compactly. Because we need 31 different values for the day, 12 different values for the month, and 100 different values for the year, we can use 5 bits to represent the day, 4 bits to represent the month, and 7 bits to represent the year. Your function should take as input the day, month, and year as integers, and it should return the date packed into a 16-bit integer. Write another function that does the unpacking. Write a program to test your functions.

17 Write a function that acts directly on a packed date, and produces the next calendar day in packed form. (See the previous exercise.) Contrast this to the program you wrote in the previous exercise.

18 Rewrite the program given in Section 4.10, "An Example: Boolean Variables," on page 170. Use the five low-order bits in the `char` variable `b` to represent the five boolean variables `b1`, ... , `b5`.

19 Rewrite the program from the previous exercise to take advantage of machine arithmetic. Show by hand simulation that the effect of adding 1 to the bit representation for `b` is equivalent to the effect of the nested `for` statements. In this exercise, your program should generate the table using a single unnested `for` statement.

20 (Balanced Meal Program) Use enumeration types to define five basic food groups: fish, fruits, grains, meats, and vegetables. Use a random number generator to select an item from each food group. Write a function `meal()` that picks an item from each of the five groups and prints out this menu. Print 20 menus. How many different menus are available?

21 Write a function that picks out five cards at random from a deck of cards. Your function should check that all the cards in the hand are distinct. Recall that the spots on a playing card that represent its numeric value are called "pips." A playing card such as the seven of hearts has a pip value 7 and a suit value hearts. The pip value for an ace is 1, a deuce is 2, ... , and a king is 13. Use enumeration types to represent the pips and suit values in your function. Write another function that prints out the hand in a visually pleasing way.

22 Write a set of routines that test whether the hand generated by the function in the previous exercise is a straight, a flush, or a full house. A straight consists of five cards that can be placed in consecutive sequence by pip value. A flush consists of five cards of the same suit. A full house is three of a kind plus a pair. Run your random hand generator and print out any hand that is one of these three kinds, along with the hand number. Continue to print out hands until one of each of the three kinds has been generated, or until you have generated 5,000 hands. If the latter happens, there is probably something wrong with your program. Do you know why?

23 In the game "paper, rock, scissors," an outcome that is not a tie is conveyed to the player by printing

```
You win.   or   You lose.
```

Rewrite the program so that messages like the following are printed:

```
You chose paper and I chose rock.  You win.
```

24 Consider the function `pack()` given in Section 7.4, "Packing and Unpacking," on page 341. The body of the function consists of four statements. Rewrite the function so that these four statements are collapsed into a single `return` statement.

25 Rewrite the function `pack()` so that only arithmetic operations are used.

26 On any machine, a mask of type `long` is acceptable. However, when we tried the following initialization on a 2-byte machine, our code did not work as expected:

```
long   mask = 1 << 31;   /* turn the high bit on: error! */
```

We made an egregious error. Can you explain what it is?

27 How multicharacter character constants such as `'abc'` get stored is system-dependent. Because programmers sometimes write `'abc'` instead of `"abc"`, some compilers provide a warning when multicharacter character constants get used, even if the use is proper. What happens on your system? Try the following code:

```
int   c = 'abc';

printf("'abc' = ");
bit_print(c);
printf("\n");
```

Here is the output on a Sun workstation:

```
'abc' = 00000000 01100011 01100010 01100001
```

28 A useful implementation of the mathematical concept of set is an `unsigned long` treated as a set of up to 32 elements.

```
typedef   unsigned long   set;

const set   empty = 0X0;      /* use hexadecimal constants */
```

Write a routine that does set union using bit operators. *Caution:* Because `union` is a keyword, use another name.

```
/* This function returns the union of a and b. */

set   Union(set a, set b);
```

By using masks you can examine whether a bit position is 1. Use this idea to write a function

```
void   display(set a);
```

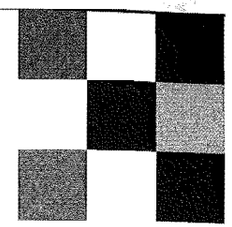

that informatively prints out the members of a set. To test your functions, you could write

```
set   a = 0X7;              /* a has elements 1, 2, 3 */
set   b = 0X55;             /* b has elements 1, 3, 5, 7 */

display(Union(a, b));       /* 1, 2, 3, 5, 7 is in the union */
```

29 (Project) Use the ideas presented in the previous exercise to develop a complete set manipulation package for sets whose size is 32 members or less. Thus, you need to write

```
set   Union(set a, set b);
set   intersection(set a, set b);
set   complement(set a);
```

After you have written these functions and tested them, use arrays of `int`s to represent larger sets. The size of the arrays should allow for sets with 1,000 members or less. Modify your functions to work with these sets.



Chapter 8

The Preprocessor

The C language uses the preprocessor to extend its power and notation. In this chapter, we present a detailed discussion of the preprocessor, including new features added by the ANSI C committee. We begin by explaining the use of `#include`. Then we thoroughly discuss the use of the `#define` macro facility. Macros can be used to generate inline code that takes the place of a function call. Their use can reduce program execution time.

Lines that begin with a # are called *preprocessing directives*. These lines communicate with the preprocessor. In ANSI C, the # can be preceded on the line by white space, whereas in traditional C, it must occur in column 1. The syntax for preprocessing directives is independent of the rest of the C language. The effect of a preprocessing directive starts at its place in a file and continues until the end of that file, or until its effect is negated by another directive. It is always helpful to keep in mind that the preprocessor does not "know C."

8.1 The Use of `#include`

We have already used preprocessing directives such as

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

Another form of the `#include` facility is given by

```
#include "filename"
```

This causes the preprocessor to replace the line with a copy of the contents of the named file. A search for the file is made first in the current directory and then in other system-dependent places. With a preprocessing directive of the form

```
#include <filename>
```

the preprocessor looks for the file only in the other places and not in the current directory. In UNIX systems, the standard header files such as *stdio.h* and *stdlib.h* are typically found in */usr/include*. In general, where the standard header files are stored is system-dependent.

There is no restriction on what a `#include` file can contain. In particular, it can contain other preprocessing directives that will be expanded by the preprocessor in turn.

8.2 The Use of #define

Preprocessing directives with `#define` occur in two forms:

```
#define   identifier   token_string(opt)
#define   identifier( identifier, ... , identifier )   token_string(opt)
```

The *token_string* is optional. A long definition of either form can be continued to the next line by placing a backslash \ at the end of the current line. If a simple `#define` of the first form occurs in a file, the preprocessor replaces every occurrence of *identifier* by *token_string* in the remainder of the file, except in quoted strings. Consider the example

```
#define   SECONDS_PER_DAY   (60 * 60 * 24)
```

In this example, the token string is `(60 * 60 * 24)`, and the preprocessor will replace every occurrence of the symbolic constant `SECONDS_PER_DAY` by that string in the remainder of the file.

The use of simple `#defines` can improve program clarity and portability. For example, if special constants such as π or the speed of light c are used in a program, they should be defined.

```
#define   PI   3.14159
#define   C    299792.458          /* speed of light in km/sec */
```


Other special constants are also best coded as symbolic constants.

```
#define   EOF      (-1)           /* typical end-of-file value */
#define   MAXINT   2147483647     /* largest 4-byte integer */
```

Program limits that are programmer decisions can also be specified symbolically.

```
#define   ITERS    50          /* number of iterations */
#define   SIZE     250         /* array size */
#define   EPS      1.0e-9      /* a numerical limit */
```

In general, symbolic constants aid documentation by replacing what might otherwise be a mysterious constant with a mnemonic identifier. They aid portability by allowing constants that may be system-dependent to be altered once. They aid reliability by restricting to one place the check on the actual representation of the constant.

Syntactic Sugar

It is possible to alter the syntax of C toward some user preference. A frequent programming error is to use the token = in place of the token == in logical expressions. A programmer could use

```
#define   EQ   ==
```

to defend against such a mistake. This superficial alteration of the programming syntax is called *syntactic sugar*. Another example of this is to change the form of the `while` statement by introducing "do," which is an ALGOL style construction.

```
#define   do   /* blank */
```

With these two `#define` lines at the top of the file, the code

```
while (i EQ 1) do {
   .....
```

will become, after the preprocessor pass,

```
while (i == 1) {
   .....
```

Keep in mind that because `do` will disappear from anywhere in the file, the `do-while` statement cannot be used. Using macros to provide a personal syntax is controversial. The advantage of avoiding == mistakes is offset by the use of an idiosyncratic style.

8.3 Macros with Arguments

So far, we have considered only simple `#define` preprocessing directives. We now want to discuss how we can use the `#define` facility to write macro definitions with parameters. The general form is given by

`#define` *identifier*`(` *identifier*`, ... ,` *identifier* `)` *token_string*$_{opt}$

There can be no space between the first identifier and the left parenthesis. Zero or more identifiers can occur in the parameter list. An example of a macro definition with a parameter is

```
#define   SQ(x)   ((x) * (x))
```

The identifier `x` in the `#define` is a parameter that is substituted for in later text. The substitution is one of string replacement without consideration of syntactic correctness. For example, with the argument `7 + w` the macro call

```
SQ(7 + w)   expands to   ((7 + w) * (7 + w))
```

In a similar fashion

```
SQ(SQ(*p))   expands to   ((((*p) * (*p))) * (((*p) * (*p))))
```

This seemingly extravagant use of parentheses is to protect against the macro expanding an expression so that it led to an unanticipated order of evaluation. It is important to understand why all the parentheses are necessary. First, suppose we had defined the macro as

```
#define   SQ(x)   x * x
```

With this definition

```
SQ(a + b)   expands to   a + b * a + b
```

which, because of operator precedence, is not the same as

```
((a + b) * (a + b))
```


Now suppose we had defined the macro as

```
#define   SQ(x)   (x) * (x)
```

With this definition

```
4 / SQ(2)   expands to   4 / (2) * (2)
```

which, because of operator precedence, is not the same as

```
4 / ((2) * (2))
```

Finally, let us suppose that we had defined the macro as

```
#define   SQ (x)   ((x) * (x))
```

With this definition

```
SQ(7)   expands to   (x)      ((x) * (x)) (7)
```

which is not even close to what was intended. If, in the macro definition, there is a space between the macro name and the left parenthesis that follows, then the rest of the line is taken as replacement text.

A common programming error is to end a `#define` line with a semicolon, making it part of the replacement string when it is not wanted. As an example of this, consider

```
#define   SQ(x)   ((x) * (x));        /* error */
```

The semicolon here was typed by mistake, one that is easily made because programmers often end a line of code with a semicolon. When used in the body of a function, the line

```
x = SQ(y);   gets expanded to   x = ((y) * (y));;
```

The last semicolon creates an unwanted null statement. If we were to write

```
if (x == 2)
   x = SQ(y);
else
   ++x;
```

we would get a syntax error caused by the unwanted null statement. The extra semicolon does not allow the `else` to be attached to the `if` statement.

Macros are frequently used to replace function calls by inline code, which is more efficient. For example, instead of writing a function to find the minimum of two values, a programmer could write

```
#define   min(x, y)   (((x) < (y)) ? (x) : (y))
```

After this definition, an expression such as

```
m = min(u, v)
```

gets expanded by the preprocessor to

```
m = (((u) < (v)) ? (u) : (v))
```

The arguments of `min()` can be arbitrary expressions of compatible type. Also, we can use `min()` to build another macro. For example, if we need to find the minimum of four values, we can write

```
#define   min4(a, b, c, d)   min(min(a, b), min(c, d))
```

A macro definition can use both functions and macros in its body. For example

```
#define   SQ(x)      ((x) * (x))
#define   CUBE(x)    (SQ(x) * (x))
#define   F_POW(x)   sqrt(sqrt(CUBE(x))) /* fractional power:3/4 */
```

A preprocessing directive of the form

```
#undef identifier
```

will undefine a macro. It causes the previous definition of a macro to be forgotten.

Caution: Debugging code that contains macros with arguments can be difficult. To see the output from the preprocessor, you can give the command

```
cc -E file.c
```

After the preprocessor has done its work, no further compilation takes place. (See exercise 1, on page 396.)

8.4 The Type Definitions and Macros in *stddef.h*

C provides the `typedef` facility so that an identifier can be associated with a specific type. A simple example is

```
typedef   char   uppercase;
```

This makes `uppercase` a type that is synonymous with `char`, and it can be used in declarations, just as other types are used. An example is

```
uppercase   c, u[100];
```

The `typedef` facility allows the programmer to use type names that are appropriate for a specific application. (See Section 9.1, "Structures," on page 408, and Section 10.6, "An Example: Polish Notation and Stack Evaluation," on page 468.)

In this section, we are concerned with the implementation-specific type definitions and macros that are given in the header file *stddef.h*. They can occur in other standard header files as well. Here is how the type definitions might appear:

```
typedef   int        ptrdiff_t;   /* pointer difference type */
typedef   short      wchar_t;     /* wide character type */
typedef   unsigned   size_t;      /* the sizeof type */
```

The type `ptrdiff_t` tells what type is obtained with an expression involving the difference of two pointers. (See exercise 7, on page 398.) The type `wchar_t` is provided to support languages with character sets that will not fit into a `char`. Some C compilers are not interested in providing such support. If that is the case, the type definition would probably be

```
typedef   char   wchar_t;     /* same as a plain char */
```

Recall that the `sizeof` operator is used to find the size of a type or an expression. For example, `sizeof(double)` on most systems is 8. The type `size_t` is the type of the result of the `sizeof` operator. This is system-dependent, but it must be an unsigned integral type. We will see the use of `size_t` in our discussion of `qsort()` in the next section.

The macro `NULL` is also given in `stddef.h`. It is an implementation-defined null pointer constant. Typically, `NULL` is defined to be 0, but on some systems it is given by

```
#define   NULL   ((void *) 0)
```

8.5 An Example: Sorting with qsort()

Programmers, for a variety of reasons, need to be able to sort data. If the amount of data is small, or the program does not have to run fast, we can use a bubble sort or a transposition sort to accomplish the task. If, however, there is a lot of data, and speed of execution is a concern, then we can use the function `qsort()` provided by the standard library. (The name *qsort* stands for "quick sort.")

The function `qsort()` is useful because it is a lot faster than a bubble sort or a simple transposition sort and it is quite easy to use. Another quicksort implementation, `quicksort()`, however, can be even faster, but it requires more coding effort. (See exercises 33 and 34 on page 406. See also Section 8.15, "An Example: Quicksort," on page 391.)

The function prototype for `qsort()` is in *stdlib.h*. It is equivalent to

```
void qsort(void *array, size_t n_els, size_t el_size,
           int compare(const void *, const void *));
```

Notice that the type `size_t` is used. Many other functions in the standard library also use this type. When `qsort()` is called, its first argument is the array to be sorted, its second argument is the number of elements in the array, its third argument is the number of bytes in an element, and its fourth argument is a function, called the *comparison function*, which is used to compare elements in the array. In this function prototype for `qsort()`, the declaration of the fourth parameter is

```
int compare(const void *, const void *)
```

This is itself a function prototype, the prototype for the comparison function. The comparison function takes as arguments two pointers to `void`. When `compare()` gets invoked, these two pointers will point to elements of the array. The comparison function returns an `int` that is less than, equal to, or greater than zero, depending on whether its first argument is considered to be less than, equal to, or greater than its second argument. The two pointers are of type `void *` because they are meant to be generic. As we will see, `qsort()` can be used to sort arrays of any type. The type qualifier `const` tells the compiler that the objects pointed to by the two pointers should not be modified. (See Section 6.19, "The Type Qualifiers `const` and `volatile`," on page 307.)

Let us write a test program that illustrates the use of `qsort()`. In our program, we fill an array, print it, sort it with `qsort()`, and then print it again.

In file try_qsort.c

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

#define   N   11                                /* size of the array */

enum when {before, after};

typedef   enum when   when;

int    cmp(const void *vp, const void *vq);
void   fill_array(double *a, int n);
void   prn_array(when val, double *a, int n);

int main(void)
{
   double   a[N];

   fill_array(a, N);
   prn_array(before, a, N);
   qsort(a, N, sizeof(double), cmp);
   prn_array(after, a, N);
   return 0;
}

int cmp(const void *vp, const void *vq)
{
   const double   *p = vp;
   const double   *q = vq;
   double          diff = *p - *q;

   return ((diff >= 0.0) ? ((diff > 0.0) ? -1 : 0) : +1);
}

void fill_array(double *a, int n)
{
   int   i;

   srand(time(NULL));                                /* seed rand() */
   for (i = 0; i < n; ++i)
      a[i] = (rand() % 1001) / 10.0;
}
```

```
void prn_array(when val, double *a, int n)
{
   int   i;

   printf("%s\n%s%s\n",
      "---",
      ((val == before) ? "Before " : "After "), "sorting:");
   for (i = 0; i < n; ++i) {
      if (i % 6 == 0)  putchar('\n');
      printf("%10.1f", a[i]);
   }
   putchar('\n');
}
```

We want to discuss a number of points about this test program, but before we do, let us look at some typical output:

```
---
Before sorting:
       1.5       17.0       99.5       45.3       52.6       66.3
       3.4       70.2       23.4       57.4        6.4
---
After sorting:
      99.5       70.2       66.3       57.4       52.6       45.3
      23.4       17.0        6.4        3.4        1.5
```

Dissection of the *try_qsort* Program

```
#define   N   11                    /* size of the array */
```

To test the function `qsort()`, we will create an array of size `N`. After we get the code to work for a small array, we can test it on a larger array.

```
int    cmp(const void *vp, const void *vq);
void   fill_array(double *a, int n);
void   prn_array(when val, double *a, int n);
```

These are function prototypes. We can name our comparison function whatever we want, but the type of the function and the number and type of its arguments must agree with the last parameter in the function prototype for `qsort()`.


```
int main(void)
{
   double   a[N];

   fill_array(a, N);
   prn_array(before, a, N);
   qsort(a, N, sizeof(double), cmp);
   prn_array(after, a, N);
   return 0;
}
```

In `main()`, we declare `a` to be an array of `double`s. Because the purpose of our program is to test `qsort()`, we do not do anything exciting. All we do is fill the array, print it, use `qsort()` to sort it, and then print the array again.

```
qsort(a, N, sizeof(double), cmp);
```

When `qsort()` is invoked, we must pass it the base address of the array to be sorted, the number of elements in the array, the number of bytes required to store an element, and the name of our comparison function.

```
int cmp(const void *vp, const void *vq)
{
   .....
```

This is the start of the function definition for our comparison function. The letter `v` in `vp` and `vq` is mnemonic for "void." In the body of `main()` we pass the name of our comparison function as the last argument to `qsort()`. This occurs in the statement

```
qsort(a, N, sizeof(double), cmp);
```

In the body of `qsort()`, which the programmer does not have access to, pointers to elements of the array `a` will be passed to `cmp()`. The programmer is not concerned with the internal details of `qsort()`. The programmer only has to write the comparison function with the understanding that the parameters `vp` and `vq` are pointers to elements of the array.

■
```
int cmp(const void *vp, const void *vq)
{
   const double   *p = vp;
   const double   *q = vq;
   double         diff = *p - *q;
   .....
```

In the body of our comparison function, we initialize `vp` to `p` and `vq` to `q`. If we do not qualify `p` and `q` with `const`, the compiler will complain. (See Section 6.19, "The Type Qualifiers `const` and `volatile`," on page 307.) The variables `p` and `q` are of type pointer to `double` because the elements of the array `a` are of type `double`. We initialize `diff` to the difference of the objects pointed to by `p` and `q`.

■
```
return ((diff >= 0.0) ? ((diff > 0.0) ? -1 : 0) : +1);
```

If `diff` is positive, we return -1; if `diff` is zero, we return 0; and if `diff` is negative, we return 1. This causes the array to be sorted in descending order. We can cause the array to be sorted in ascending order if we replace this line with

```
if (diff < 0.0)
   return -1;
if (diff == 0.0)
   return 0;
return 1;
```

■
```
void fill_array(double *a, int n)
{
   int   i;

   srand(time(NULL));                  /* seed rand() */
   .....
```

Typically, the function call `time(NULL)` returns the number of seconds that have elapsed since 1 January 1970. ANSI C does not guarantee this, but this convention is widely followed. Passing `time(NULL)` to `srand()` causes the array `a` to be filled with different values every time the program is invoked.

■
```
for (i = 0; i < n; ++i)
   a[i] = (rand() % 1001) / 10.0;
```

The expression `rand() % 1001` has an `int` value in the interval 0 to 1000. Because we are dividing this by the `double` value 10.0, the value of what is assigned to `a[i]` is in the interval 0 to 100.

■ ■ ■ ■

8.6 An Example: Macros with Arguments

In this section, we again fill arrays and sort them with `qsort()`, but this time we use macros with arguments. We will call our program *sort*.

Let us write our program in three files: a header file *sort.h* and two *.c* files. In the header file we put our `#includes`, our `#defines`, and a list of function prototypes.

In file sort.h

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

#define   M   32                                /* size of a[] */
#define   N   11                                /* size of b[] */
#define   fractional_part(x)   (x - (int) x)
#define   random_char()        (rand() % 26 + 'a')
#define   random_float()       (rand() % 100 / 10.0)

#define   FILL(array, sz, type)     \
   if (strcmp(type, "char") == 0)   \
      for (i = 0; i < sz; ++i)      \
         array[i] = random_char();  \
   else                             \
      for (i = 0; i < sz; ++i)      \
         array[i] = random_float()

#define   PRINT(array, sz, cntrl_string)  \
   for (i = 0; i < sz; ++i)               \
      printf(cntrl_string, array[i]);     \
   putchar('\n')

int   compare_fractional_part(const void *, const void *);
int   lexico(const void *, const void *);
```

Dissection of the *sort.h* Header File

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

The header file *stdio.h* contains the macro definition for NULL and the function prototype for `printf()`. The header file *stdlib.h* contains the function prototypes for `rand()`, `srand()`, and `qsort()`. The header file *time.h* contains the function prototype for `time()`. The function call `time(NULL)` will be used to seed the random number generator.

```
#define   fractional_part(x)   (x - (int) x)
```

If `x` is a positive `float`, then the expression `x - (int) x` yields the fractional part of `x`.

```
#define   random_char()        (rand() % 26 + 'a')
```

When `rand()` is invoked, it returns an integer value randomly distributed between 0 and `MAX_RAND`, a symbolic constant defined in *stdlib.h*. Because `MAX_RAND` is typically more than 32 thousand, the expression `rand() % 26` yields an integer value randomly distributed between 0 and 25. Because `0 + 'a'` has the value `'a'` and `25 + 'a'` has the value `'z'`, the expression

```
rand() % 26 + 'a'
```

produces a character value randomly distributed between `'a'` and `'z'`.

```
#define   random_float()       (rand() % 100 / 10.0)
```

The value of the expression `rand() % 100` is an integer randomly distributed between 0 and 99. Because the expression `10.0` is of type `double`, the value produced by `rand() % 100` is promoted to a `double`, and the expression

```
rand() % 100 / 10.0
```

as a whole is also of type `double`. Its value is between 0 and 9.9.


```
#define   FILL(array, sz, type)     \
   if (strcmp(type, "char") == 0)   \
      for (i = 0; i < sz; ++i)      \
         array[i] = random_char();  \
   else                             \
      for (i = 0; i < sz; ++i)      \
         array[i] = random_float()
```

In this macro definition, `array`, `sz`, and `type` are parameters. Unlike function definitions, no type checking gets done. It is the programmer's responsibility to call the macro with arguments of the appropriate type. Note that the variable `i` is used in the body of the macro. Because it is not declared here, it has to be declared in `main()`, where the macro gets called. Consider the macro call

```
FILL(a, n, "char");
```

When the macro gets expanded, we obtain

```
if (strcmp("char", "char") == 0)
   for (i = 0; i < n; ++i)
      a[i] = random_char();
else
   for (i = 0; i < n; ++i)
      a[i] = random_float();
```

The identifiers `array`, `sz`, and `type` have been replaced by `a`, `n`, and `"char"`, respectively. Note carefully that all but the last semicolon came from the preprocessor expansion mechanism.

```
#define   PRINT(array, sz, cntrl_string)  \
   for (i = 0; i < sz; ++i)               \
      printf(cntrl_string, array[i]);     \
   putchar('\n')
```

This macro can be used to print the values of elements of an array. Note that the control string for `printf()` is a parameter in the macro definition.

```
int   compare_fractional_part(const void *, const void *);
int   lexico(const void *, const void *);
```

These are prototypes of comparison functions that will be passed to `qsort()`. Notice that with respect to type, they match the function prototype of the comparison function in the function prototype for `qsort()`.

Now let us consider the rest of the code for our program. In `main()`, we fill an array, print it, sort it, and print it again. Then we repeat the process, but this time with an array of a different type.

In file main.c

```
#include "sort.h"

int main(void)
{
   char    a[M];
   float   b[N];
   int     i;

   srand(time(NULL));
   FILL(a, M , "char");
   PRINT(a, M, "%-2c");
   qsort(a, M, sizeof(char), lexico);
   PRINT(a, M, "%-2c");
   printf("---\n");
   FILL(b, N, "float");
   PRINT(b, N, "%-6.1f");
   qsort(b, N, sizeof(float), compare_fractional_part);
   PRINT(b, N, "%-6.1f");
   return 0;
}
```

Notice that each time we invoke `qsort()`, we use a different comparison function. Here is the output of our program:

```
q m z r h l a j o e t b k w l t z t v i e m h p f y b p s w a j
a a b b e e f h h i j j k l l m m o p p q r s t t t v w w y z z
---
9.4   0.2   5.1   6.7   5.4   5.3   6.1   9.6   2.8   8.8   8.5
6.1   5.1   0.2   5.3   5.4   9.4   8.5   9.6   6.7   8.8   2.8
```

Finally, we want to look at the two comparison functions. Pointers to `void` are used because this is required by the function prototype of `qsort()` in *stdlib.h*. We will carefully explain how these pointers get used in the comparison functions.

In file compare.c

```
#include "sort.h"

int compare_fractional_part(const void *vp, const void *vq)
{
   const float   *p = vp, *q = vq;
   float         x;

   x = fractional_part(*p) - fractional_part(*q);
   return ((x < 0.0) ? -1 : (x == 0.0) ? 0 : +1);
}

int lexico(const void *vp, const void *vq)
{
   const char   *p = vp, *q = vq;
   return (*p - *q);
}
```

Dissection of the compare_fractional_part() Function

```
int compare_fractional_part(const void *vp, const void *vq)
{
   .....
```

This function takes two `const` qualified pointers to `void` as arguments and returns an `int`. Because of this, the function can be passed as an argument to `qsort()`.

```
int compare_fractional_part(const void *vp, const void *vq)
{
   const float   *p = vp, *q = vq;
   float         x;
   .....
```

The letter `v` in `vp` and `vq` is mnemonic for "void." Because pointers to `void` cannot be dereferenced, we declare `p` and `q` to be pointers to `float` and initialize them with `vp` and `vq`, respectively. Because an ANSI C compiler will complain if a `const`-qualified pointer is assigned to one that is not `const` qualified, we declare `p` and `q` to be `const` qualified. Notice that we did not declare `x` to be `const` qualified. If we had done so, we would be able to give `x` a value only by initializing it.

```
    x = fractional_part(*p) - fractional_part(*q);
    return ((x < 0.0) ? -1 : (x == 0.0) ? 0 : +1);
```

The difference of the fractional parts of the objects pointed to by p and q is assigned to x. Then -1, 0, or +1 is returned, depending on whether x is negative, zero, or positive. Thus, when we call qsort() with compare_decimal_part() passed as an argument, the elements in the array get sorted according to their fractional parts.

Observe that in the function lexico() we defined p and q to be pointers to const char and initialized them with vp and vq, respectively. Then we returned the difference of what is pointed to by p and q. Thus, when we call qsort() with lexico passed as an argument, the elements in the array get sorted lexicographically.

8.7 The Macros in *stdio.h* and *ctype.h*

The C system provides the macros getc() and putc() in *stdio.h*. The first is used to read a character from a file, the second to write a character to a file. (See Section 11.5, "An Example: Double Spacing a File," on page 509.) Because the header file *stdio.h* contains the lines

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

we see that getchar() and putchar() are also macros. They read characters from the keyboard and write characters to the screen, respectively.

The C system also provides the standard header file *ctype.h*, which contains a set of macros that test characters and a set of prototypes of functions that convert characters. The preprocessing directive

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

includes these macros and function prototypes. In the table that follows, we list the macros that test characters. These macros all take an argument of type int and return an int.

Macro	Nonzero (true) is returned if:
isalpha(c)	c is a letter
isupper(c)	c is an uppercase letter
islower(c)	c is a lowercase letter
isdigit(c)	c is a digit
isalnum(c)	c is a letter or digit
isxdigit(c)	c is a hexadecimal digit
isspace(c)	c is a white space character
ispunct(c)	c is a punctuation character
isprint(c)	c is a printable character
isgraph(c)	c is printable, but not a space
iscntrl(c)	c is a control character
isascii(c)	c is an ASCII code

In the next table, we list the functions toupper() and tolower(), which are in the standard library, and the macro toascii(). The macro and the prototypes for the two functions are in *ctype.h*. The functions and the macro each take an int and return an int. In the table, we assume that c is a variable of integral type, such as char or int. Note carefully that the value of c stored in memory does not get changed.

Call to the function or macro	Value returned
toupper(c)	corresponding uppercase value or c
tolower(c)	corresponding lowercase value or c
toascii(c)	corresponding ASCII value

If c is not a lowercase letter, then the value returned by toupper(c) is c. Similarly, if c is not an uppercase letter, then the value returned by tolower(c) is c.

8.8 Conditional Compilation

The preprocessor has directives for conditional compilation. They can be used for program development and for writing code that is more easily portable from one machine to another. Each preprocessing directive of the form

```
#if       constant_integral_expression
#ifdef    identifier
#ifndef   identifier
```

provides for conditional compilation of the code that follows until the preprocessing directive

```
#endif
```

is reached. For the intervening code to be compiled, after `#if` the constant expression must be nonzero (*true*), and after `#ifdef` or after `#if defined`, the named identifier must have been defined previously in a `#define` line, without an intervening

`#undef` *identifier*

having been used to undefine the macro. After `#ifndef`, the named identifier must be currently undefined.

The integral constant expression used in a preprocessing directive cannot contain the `sizeof` operator or a cast, but it may use the `defined` preprocessing operator. This operator is available in ANSI C, but not necessarily in traditional C. The expression

`defined` *identifier* is equivalent to `defined(`*identifier*`)`

It evaluates to 1 if the identifier is currently defined, and evaluates to 0 otherwise. Here is an example of how it can be used:

```
#if defined(HP9000) || defined(SUN4) && !defined(VAX)
   .....      /* machine-dependent code */
#endif
```

Sometimes `printf()` statements are useful for debugging purposes. Suppose that at the top of a file we write

```
#define   DEBUG   1
```


and then throughout the rest of the file we write lines such as

```
#if DEBUG
   printf("debug: a = %d\n", a);
#endif
```

Because the symbolic constant `DEBUG` has nonzero value, the `printf()` statements will be compiled. Later, these lines can be omitted from compilation by changing the value of the symbolic constant `DEBUG` to 0.

An alternate scheme is to define a symbolic constant having no value. Suppose that at the top of a file we write

```
#define   DEBUG
```

Then we can use the `#ifdef` or `#if defined` forms of conditional compilation. For example, if we write

```
#ifdef DEBUG
   .....
#endif
```

then the intervening lines of code will be compiled. When we remove the `#define` line that defines `DEBUG` from the top of the file, the intervening lines of code will not be compiled.

Suppose we are writing code in a large software project. We may be expected to include at the top of all our code certain header files supplied by others. Our code may depend on some of the function prototypes and on some of the macros in these header files, but because the header files are for the project as a whole, our code might not use everything. Moreover, we may not even know all the things that eventually will be in the header files. To prevent the clash of macro names, we can use the `#undef` facility:

```
#include "everything.h"

#undef    PIE
#define   PIE   "I like apple."
.....
```

If `PIE` happens to be defined in *everything.h*, then we have undefined it. If it is not defined in *everything.h*, then the `#undef` directive has no effect.

Here is a common use of conditional compilation. Imagine that you are in the testing phase of program development and that your code has the form

```
statements
more statements
and still more statements
```

For debugging or testing purposes, you may wish to temporarily disregard, or block out, some of your code. To do this, you can try to put the code into a comment.

```
statements
/*
more statements
*/
and still more statements
```

However, if the code to be blocked out contains comments, this method will result in a syntax error. The use of conditional compilation solves this problem.

```
statements
#if 0
more statements
#endif
and still more statements
```

The preprocessor has control structures that are similar to the `if-else` statement in C. Each of the `#if` forms can be followed by any number of lines, possibly containing preprocessing directives of the form

```
#elif    constant_integral_expression
```

possibly followed by the preprocessing directive

```
#else
```

and, finally, followed by the preprocessing directive

```
#endif
```

Note that `#elif` is a contraction for "else-if." The flow of control for conditional compilation is analogous to that provided by `if-else` statements.

8.9 The Predefined Macros

In ANSI C, there are five predefined macros. They are always available, and they cannot be undefined by the programmer. Each of these macro names includes two leading and two trailing underscore characters.

Predefined macro	Value
`__DATE__`	A string containing the current date
`__FILE__`	A string containing the file name
`__LINE__`	An integer representing the current line number
`__STDC__`	If the implementation follows ANSI Standard C, then the value is a nonzero integer.
`__TIME__`	A string containing the current time

8.10 The Operators # and

The preprocessing operators # and ## are available in ANSI C but not in traditional C. The unary operator # causes "stringization" of a formal parameter in a macro definition. Here is an example of its use:

```
#define   message_for(a, b)  \
          printf(#a " and " #b ": We love you!\n")

int main(void)
{
   message_for(Carole, Debra);
   return 0;
}
```

When the macro is invoked, each parameter in the macro definition is replaced by its corresponding argument, with the # causing the argument to be surrounded by double quotes. Thus, after the preprocessor pass, we obtain

```
int main(void)
{
   printf("Carole" " and " "Debra" ": We love you!\n");
   return 0;
}
```

Because string constants separated by white space are concatenated, this `printf()` statement is equivalent to

```
printf("Carole and Debra: We love you!\n");
```

In the next section we use the "stringization" operator # in the `assert()` macro.

The binary operator ## is used to merge tokens. Here is an example of how the operator is used:

```
#define   X(i)   x ## i
X(1) = X(2) = X(3);
```

After the preprocessor pass, we are left with the line

```
x1 = x2 = x3;
```

8.11 The assert() Macro

ANSI C provides the `assert()` macro in the standard header file *assert.h*. This macro can be used to ensure that the value of an expression is what you expect it to be. Suppose that you are writing a critical function and that you want to be sure the arguments satisfy certain conditions. Here is an example of how `assert()` can be used to do this:

```
#include <assert.h>

void f(char *p, int n)
{
   assert(p != NULL);
   assert(n > 0 && n < 7);
   .....
```

If an assertion fails, then the system will print out a message and abort the program. Although the `assert()` macro is implemented differently on each system, its general behavior is always the same. Here is one way the macro might be written:


```
#include <stdio.h>
#include <stdlib.h>                              /* for abort() */

#if defined(NDEBUG)
   #define   assert(ignore)   ((void) 0)         /* ignore it */
#else
   #define   assert(expr)                       \
      if (!(expr)) {                            \
       printf("\n%s%s\n%s%s\n%s%d\n\n",         \
          "Assertion failed: ", #expr,          \
          "in file ", __FILE__,                 \
          "at line ", __LINE__);                \
       abort();                                 \
      }
#endif
```

Note that if the macro `NDEBUG()` is defined, then all assertions are ignored. This allows the programmer to use assertions freely during program development, and then to effectively discard them later by defining the macro `NDEBUG`. The function `abort()` is in the standard library. (See Appendix A, "The Standard Library.")

8.12 The Use of #error and #pragma

ANSI C has added the `#error` and `#pragma` preprocessing directives. The following code demonstrates how `#error` can be used:

```
#if A_SIZE < B_SIZE
   #error "Incompatible sizes"
#endif
```

If during compilation the preprocessor reaches the `#error` directive, then a compile-time error will occur, and the string following the directive will be printed on the screen. In our example, we used the `#error` macro to enforce the consistency of two symbolic constants. In an analogous fashion, the directive can be used to enforce other conditions. The `#pragma` directive is provided for implementation-specific uses. Its general form is

```
#pragma    tokens
```

It causes a behavior that depends on the particular C compiler. Any `#pragma` that is not recognized by the compiler is ignored.

8.13 Line Numbers

A preprocessing directive of the form

```
#line   integral_constant   "filename"
```

causes the compiler to renumber the source text so the next line has the specified constant and causes it to believe that the current source file name is *filename*. If no file name is present, then only the renumbering of lines takes place. Normally, line numbers are hidden from the programmer and occur only in reference to warnings and syntax errors.

8.14 Corresponding Functions

In ANSI C, many of the macros with parameters that are given in the standard header files also have corresponding functions in the standard library. As an example, suppose we want to access the function `isalpha()` instead of the macro. One way to do this is to write

```
#undef isalpha
```

somewhere in the file before `isalpha()` is invoked. This has the effect of discarding the macro definition, forcing the compiler to use the function instead. We would still include the header file *ctype.h* at the top of the file, however, because in addition to macros, the file contains function prototypes.

Another way to obtain the function instead of the macro is to write

```
(isalpha)(c)
```

The preprocessor does not recognize this construct as a macro, but the compiler recognizes it as a function call. (See exercise 8, on page 398.)

8.15 An Example: Quicksort

Quicksort was created by C. Anthony R. Hoare and described in his 1962 paper "Quicksort" (*Computer Journal*, vol. 5, no. 1). Of all the various sorting techniques, quicksort is perhaps the most widely used internal sort. An internal sort is one in which all the data to be sorted fit entirely within main memory.

Our quicksort code makes serious use of macros. When sorting lots of data, speed is essential, and the use of macros instead of functions helps to make our code run faster. As we will see, all the macros that we use are quite simple. We could replace them by inline code, but their use makes the code more readable. Because quicksort is important, we will explain it in some detail.

Let us suppose that we want to sort an array of integers of size n. If the values of the elements are randomly distributed, then, on average, the number of comparisons done by quicksort is proportional to $n \log n$. But in the worst case, the number of comparisons is proportional to n^2. This is a disadvantage of quicksort. Other sorts—mergesort, for example—even in the worst case, do work proportional to $n \log n$. However, of all the $n \log n$ sorting methods known, quicksort is, on average, the fastest by a constant factor. Another advantage of quicksort is that it does its work in place. No additional work space is needed.

Our quicksort code is written in a single file. We will describe the elements of the code as we present it. At the top of the file we have

```
/* Quicksort!  Pointer version with macros. */

#define   swap(x, y)     { int t; t = x; x = y; y = t; }
#define   order(x, y)    if (x > y) swap(x, y)
#define   o2(x, y)       order(x, y)
#define   o3(x, y, z)    o2(x, y); o2(x, z); o2(y, z)

typedef   enum {yes, no}   yes_no;

static yes_no   find_pivot(int *left, int *right, int *pivot_ptr);
static int      *partition(int *left, int *right, int pivot);
```

We have not written our macros to be robust. They are intended for use only in this file. A `typedef` has been used to make the type `yes_no` synonymous with the enumeration type `enum {yes, no}`. Because the two functions `find_pivot()` and `partition()` have `static` storage class, they are known only in this file.

```
void quicksort(int *left, int *right)
{
   int   *p, pivot;

   if (find_pivot(left, right, &pivot) == yes) {
      p = partition(left, right, pivot);
      quicksort(left, p - 1);
      quicksort(p, right);
   }
}
```

Quicksort is usually implemented recursively. The underlying idea is to "divide and conquer." Suppose that in `main()` we have declared a to be an array of size `N`. After the array has been filled, we can sort it with the call

```
quicksort(a, a + N - 1);
```

The first argument is a pointer to the first element of the array; the second argument is a pointer to the last element of the array. In the function definition for `quicksort()`, it is convenient to think of these pointers as being on the left and right side of the array, respectively. The function `find_pivot()` chooses, if possible, one of the elements of the array to be a "pivot element." The function `partition()` is used to rearrange the array so that the first part consists of elements all of whose values are less than the pivot and the remaining part consists of elements all of whose values are greater than or equal to the pivot. In addition, `partition()` returns a pointer to an element in the array. Elements to the left of the pointer all have value less than the pivot, and elements to the right of the pointer, as well as the element pointed to, all have value greater than or equal to the pivot. Once the array has been rearranged with respect to the pivot, `quicksort()` is invoked on each subarray.

```
static yes_no find_pivot(int *left, int *right, int *pivot_ptr)
{
   int   a, b, c, *p;

   a = *left;                                    /* left value */
   b = *(left + (right - left) / 2);          /* middle value */
   c = *right;                                  /* right value */
   o3(a, b, c);                          /* order these 3 values */
   if (a < b) {          /* pivot will be higher of 2 values */
      *pivot_ptr = b;
      return yes;
   }
```



```
   if (b < c) {
      *pivot_ptr = c;
      return yes;
   }
   for (p = left + 1; p <= right; ++p)
      if (*p != *left) {
       *pivot_ptr = (*p < *left) ? *left : *p;
       return yes;
      }
   return no;            /* all elements have the same value */
}
```

Ideally, the pivot should be chosen so that at each step the array is partitioned into two parts, each with an equal (or nearly equal) number of elements. This would minimize the total amount of work performed by `quicksort()`. Because we do not know *a priori* what this value should be, we try to select for the pivot the middle value from among the first, middle, and last elements of the array. In order for there to be a partition, there has to be at least one element that is less than the pivot. If all the elements have the same value, a pivot does not exist and `no` is returned by the function. (See exercise 25, on page 404 and exercise 26, on page 404, for further discussion.)

```
static int *partition(int *left, int *right, int pivot)
{
   while (left <= right) {
      while (*left < pivot)
       ++left;
      while (*right >= pivot)
       --right;
      if (left < right) {
       swap(*left, *right);
       ++left;
       --right;
      }
   }
   return left;
}
```

The major work is done by `partition()`. We want to explain in detail how this function works. Suppose we have an array `a[]` of 12 elements:

```
7   4   3   5   2   5   8   2   1   9   -6   -3
```

When `find_pivot()` is invoked, the first, middle, and last elements of the array are compared. The middle value is 5, and because this is larger than the smallest of the three values, this value is chosen for the pivot value. The following table shows the val-

ues of the elements of the array after each pass of the outer `while` loop in the `partition()` function. The elements that were swapped in that pass are boxed.

```
Unordered data:  7   4   3   5   2   5   8   2   1   9  -6  -3
First pass:     -3   4   3   5   2   5   8   2   1   9  -6   7
Second pass:    -3   4   3  -6   2   5   8   2   1   9   5   7
Third pass:     -3   4   3  -6   2   1   8   2   5   9   5   7
Fourth pass:    -3   4   3  -6   2   1   2   8   5   9   5   7
```

Notice that after the last pass the elements with index 0 to 6 have value less than the pivot and that the remaining elements have value greater than or equal to the pivot. The address of `a[7]` is returned when the function exits.

Summary

1 The preprocessor provides facilities for file inclusion and for defining macros. Files may be included by using preprocessing directives of the form

`#include` <*filename*> `#include` "*filename*"

2 A `#define` preprocessing directive can be used to give a symbolic name to a token string. The preprocessor substitutes the string for the symbolic name in the source text before compilation.

3 The use of the `#define` facility to define symbolic constants enhances readability and portability of programs.

4 The preprocessor provides a general macro facility with argument substitution. A macro with parameters is defined by a preprocessing directive of the form

`#define` *identifier*`(` *identifier*`, ..., ` *identifier* `)` $token\_string_{opt}$

An example is given by

```
#define  SQ(X)  ((X) * (X))
```

This macro provides inline code to perform squaring of a value.

5 The preprocessor provides for conditional compilation to aid in program testing, to facilitate porting, and so on. Lines beginning with `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, and `#endif` are used for this.

6 The `defined` operator can be used with preprocessing directives. An example is

```
#if (defined(HP3000) || defined(SUN3)) && !defined(SUN4)
   .....     /* machine-dependent code */
#endif
```

7 An effective way to block out sections of code for debugging purposes is to use

```
#if 0

.....
#endif
```

8 ANSI C has introduced the preprocessing operators # and ##. The # operator is unary. It can be applied to a formal parameter in the body of a macro, causing replacement text to be surrounded by double quotes. This effect is called "stringization." The ## operator is binary. It causes the pasting together of two tokens.

9 ANSI C provides the `assert()` macro in the header file *assert.h*. Assertions can be used to ensure that an expression has an appropriate value.

10 The function `qsort()` is provided by the C system. Its function prototype is in *stdlib.h*. Although `qsort()` is faster than a bubble sort or a simple transposition sort, it is often not as fast as quicksort. An advantage of `qsort()` is that it is easy to use.

11 Quicksort is one of the most widely used sorting algorithms. It is typically faster by a constant factor than other well-known $n \log n$ sorting methods, such as mergesort.

Exercises

1 A program that contains macros with arguments can be difficult to debug. Most C compilers provide an option that causes the preprocessor to write its output on the screen with no further compilation taking place. Put the following code in a file, say *try_me.c*:

```
#include <stdio.h>

#define   PRN(x)   printf("x\n");

int main(void)
{
   PRN(Hello from main());
   return 0;
}
```

Next, compile the program and run it. You will see that it does not print what was expected. To see how the preprocessor treats this code, give the command

```
cc -E try_me.c
```

(Use redirection if you want to take a careful look at what gets produced.) If the *-E* option is not the right one for your compiler, find out what the correct option is. Note that the identifier `PRN` does not get generated by the preprocessor. Explain why. Fix the code. *Hint:* Use stringization.

2 Consider the following macro definition:

```
#define   forever(x)   forever(forever(x))
forever(more)
```

This looks like it will produce infinite recursion, but in ANSI C the preprocessor is supposed to be smart enough to know that infinite recursion is not what is intended. How does your preprocessor expand this macro?

3 Suppose that `x`, `y`, and `z` are variables of type `float` in a program. If these variables have the values 1.1, 2.2, and 3.3, respectively, then the statement

```
PRN3(x, y, z);
```


should cause the line

```
x has value 1.1 and y has value 2.2 and z has value 3.3
```

to be printed. Write the macro definition for `PRN3()`.

4 Suppose we have the following code in a file named *a_b_c.h:*

```
#define   TRUE    1
#define   A_B_C   int main(void)                          \
                  {                                       \
                      printf("A Big Cheery \"hello\"!\n"); \
                      return 0;                           \
                  }
```

and the following code in a file named *a_b_c.c*:

```
#if TRUE
   #include <stdio.h>
   #include "a_b_c.h"
   A_B_C
#endif
```

When we try to compile the program, the compiler complains. Why? Can you interchange two lines in one of the files so that the program compiles and runs? Explain.

5 Macros are not always as safe as functions, even when all the parameters in the body of the macro definition are enclosed in parentheses. Define a macro

```
MAX(x, y, z)
```

that produces a value corresponding to the largest of its three arguments. Construct some expressions to use in `MAX()` that produce unanticipated results.

6 Are all of the predefined macros available on your system? Try the following code:

```
printf("%s%s\n%s%s\n%s%d\n%s%d\n%s%s\n",
   "__DATE__ = ", __DATE__,
   "__FILE__ = ", __FILE__,
   "__LINE__ = ", __LINE__,
   "__STDC__ = ", __STDC__,
   "__TIME__ = ", __TIME__);
```

7 Do you have access to an ANSI C compiler on a small system? On small systems, compilers such as Borland C and Microsoft C provide for different memory models, and each memory model usually requires a specific type definition for `ptrdiff_t`. Look in *stddef.h* and see if this is the case on your small system. Can you explain why the different memory models require their own `ptrdiff_t` type?

8 In ANSI C, many of the macros with arguments defined in the standard header files are required to be available also as functions. Does your system provide these functions? See, for example, if your system will accept the code

```
#include <ctype.h>      /* function prototype here? */

if ((isalpha)('a'))
   printf("Found the isalpha() function!\n");
```

Do not be too surprised if your system does not provide the corresponding functions. After all, they do not get used very much.

9 C has the reputation for being an excellent language for character processing. This reputation is due, in part, to the fact that macros rather than functions are used extensively in character processing. Programmers believe that the use of macros can reduce execution time significantly. Is this really true? In this exercise, we want to test this belief. Begin by writing a program that uses the macros in *stdio.h* and *ctype.h* extensively.

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

int main(void)
{
   int   c;

   printf("Clock ticks: %ld\n", clock());   /* start the clock */
   while ((c = getchar()) != EOF)
      if (islower(c))
       putchar(toupper(c));
      else if (isupper(c))
       putchar(tolower(c));
      else if (isdigit(c))
       .....
```

Complete this program. (Read about `clock()` in Section A.15, "Date and Time: <time.h>," on page 675.) If `c` is a digit, write the character `x`; if `c` is a punctuation character, do not write anything; if `c` is a white space character, write it twice. Just before the return from `main()`, write the line


```
printf("Clock ticks: %ld\n", clock());   /* ticks up to now */
```

Now, write another version of your program in which each macro is replaced by its corresponding function. For example, instead of `islower(c)` write `(islower)(c)`. Use redirection to process character files with each version of your program. On a small file, due to system overhead, there should not be much difference in running time, but as the file gets larger, the difference should be more pronounced. Is it? *Hint:* Give the command

```
pgm < input > output
```

so that you will not have to waste time printing the file on the screen.

10 In ANSI C, the standard header files can be included repeatedly and in any order. Change the first version of the program that you wrote in the previous exercise by duplicating the `#include` lines at the top of the file a number of times:

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

.....                          /* repeat a number of times */
int main(void)
{
   .....
```

Does your compiler complain? Although the program might compile a little more slowly, it should execute just as fast. Does it?

11 We listed `isascii()` as a macro in *ctype.h*. However, this macro is not really specified in the ANSI C documents. (Perhaps the ANSI C committee did not want to show any favoritism for ASCII codes over any other.) Check to see if `isascii()` is provided by your system.

12 In this exercise, we want to warn you about a subtle difference between traditional C and ANSI C. In traditional C, `tolower()` and `toupper()` are provided as macros in *ctype.h*. In ANSI C, the corresponding macros are available in *ctype.h*, but they have been renamed as `_tolower()` and `_toupper()`. (Check to see that they are available on your system.) In traditional C, it makes sense to use an expression such as `toupper(c)` only if you already know that `c` is a lowercase letter. In ANSI C, `toupper()` is implemented as a function, and the expression `toupper(c)` makes sense no matter what integral value `c` might have. If `c` is not a lowercase letter, then `toupper(c)` has no effect; that is, it returns `c`. Similar remarks hold with respect to `tolower()`. Experiment with your system. See what is produced by the expressions

```
tolower('a')    _tolower('a')    toupper('A')    _toupper('A')
```

13 The stringization operator # causes an argument that is passed to a macro to be surrounded by double quotes. What happens if the argument is already surrounded by double-quote characters? Write a test program that contains the following code:

```
#define   YANK(x)   s = #x
char   *s;
YANK("Go home, Yankee!");
printf("%s\n", s);
```

Write another version of your program that does not contain the # operator. Execute both versions. How does the output of one version differ from the other? Use the *-E* option (or whatever is required by your system) to get the preprocessor to expand your code. What does the preprocessor do differently when the # operator is present?

14 What gets printed? Explain.

```
#define   GREETINGS(a, b, c) \
          printf(#a ", " #b ", and " #c ": Hello!\n")

int main(void)
{
   GREETINGS(Alice, Bob, Carole);
   return 0;
}
```

Look what is produced by the preprocessor before compilation. Can you find `GREETINGS`?

15 Consider the directive

```
#undef   TRY_ME
```

If `TRY_ME` was previously defined with a `#define` macro, this line causes the macro to be discarded. If `TRY_ME` was not previously defined, then the line should have no effect. Write some code to test what happens on your system. If `TRY_ME` was not previously defined, does your system complain?

16 The `assert()` macro is supposed to be discarded if the macro `NDEBUG` is defined. Does it work as expected on your system? Try the following program:

```
#define   NDEBUG

#include <assert.h>

int main(void)
{
   int   a = 1, b = 2;

   assert(a > b);
   return 0;
}
```

What happens if you interchange the first two lines of the program?

17 In the program in the previous exercise, replace the two lines at the top with the following three lines:

```
#include <assert.h>

#define   NDEBUG

#include <assert.h>
```

Does your C compiler complain? Should your C compiler complain? *Hint:* Check to see if the line

```
#undef   assert
```

is in the *assert.h* header file.

18 Suppose you are moving a large C program from a machine that has 4-byte words to one that has 2-byte words. On a machine with 2-byte words, an `int` is restricted to values that lie (approximately) between -32,000 and +32,000. Suppose that this range of values is too restrictive for some parts of the program that you are moving. If you put the line

```
#define   int   long
```

into a header file that gets included with each of the files making up your program, will this work? Explain.

19 Find the `typedef` for `size_t` in the header file *stddef.h* on your system. Search for this `typedef` in *stdlib.h* also. Suppose you are writing some code that starts with

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

Because duplicate type definitions do not work, your system must be able to prevent the `typedef` for `size_t` from being included twice. Explain how this mechanism works.

20 If you want to use `qsort()`, you have to know what its function prototype is. On some ANSI C systems it is given in *stdlib.h* as

```
void qsort(void *base, size_t nelem, size_t width,
           int (*compare)(const void *, const void *));
```

What is provided by your system? Is it equivalent to this? Remember, in a function prototype, the parameter identifiers are discarded by the compiler. Thus, an equivalent function prototype is given by

```
void qsort(void *, size_t, size_t, int (*)());
```

21 In the *qsort* program, we used two comparison functions. What happens if you rewrite the comparison functions in such a way that they do not match the last parameter in the function prototype for `qsort()`? Rewrite them as

```
int compare_decimal_part(float *p, float *q)
{
   float   x;

   x = decimal_part(*p) - decimal_part(*q);
   return ((x == 0.0) ? 0 : (x < 0.0) ? -1 : +1);
}
```



```
int lexico(char *p, char *q)
{
   return (*p - *q);
}
```

Also, change the corresponding function prototypes in `main()`. Will the program compile now? If it does, take a careful look at the function prototype for `qsort()` as given in *stdlib.h* on your system. Is it the case that the last parameter is something like `(compare *)()`? If the program does not compile, can you cast the last argument in the two calls to `qsort()` so that it does?

22 Write a program to test the quicksort code. Begin with the following code:

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

#define   N   10000

void   quicksort(int *, int *);

int main(void)
{
   int   a[N], i;

   srand(time(NULL));      /* seed the random number generator */
   for (i = 0; i < N; ++i)
      a[i] = rand() % 1000;
   quicksort(a, a + N - 1);
   .....
```

Complete this program by writing code that prints the sorted array elements. If you get tired of looking at all those elements, you can print just enough of the beginning, middle, and end of the array so that you believe it has been sorted.

23 In the previous exercise, you sorted an array of size 10,000 with integer entries that were randomly distributed in the interval [0, 999]. Run the program again, this time keeping track of its running time. Now change the program so that the entries in the array are randomly distributed in the interval [0, 9]. Give a heuristic argument to explain why the times are so different.

24 The quicksort algorithm can be used to sort all kinds of arrays, not just arrays of integers. Rewrite the quicksort code so that it can be used to sort an array of strings. Write a program to test your code.

25 If an array has just a few elements, say seven or less, then a bubble sort or a transposition sort should be faster than quicksort. The following version of `quicksort()` takes this into account:

```
int quicksort(int *left, int *right)
{
   int          *p, *q, pivot;
   static int   cnt = 0;

   if (right - left < 7) {
      for (p = left; p < right; ++p)
         for (q = p + 1; q <= right; ++q)
            if (*p > *q)
               swap(*p, *q);
   }
   else if (find_pivot(left, right, &pivot) == yes) {
      p = partition(left, right, pivot);
      quicksort(left, p - 1);
      quicksort(p, right);
   }
   return ++cnt;
}
```

Note the use of the variable `cnt`. The value returned to the calling environment is the number of times that the function gets called. Experiment to see if this new version of `quicksort()` executes faster. Is there a correlation between running time and the number of times the function gets called?

26 Having a good algorithm for finding the pivot element in quicksort can be crucial. A simple algorithm is to find two distinct elements of the array and to choose the larger value as the pivot. With this algorithm, quicksort takes time proportional to n^2 instead of $n \log n$ if the array happens to be already in order, or nearly so. This is an important point, because, in practice, arrays are often partially in order to begin with. Rewrite the quicksort code to implement this algorithm and write a test program that illustrates the poor behavior of quicksort when the array is in order to begin with. What happens if the array starts in reverse order?

27 To find the pivot element, we choose from among three elements of the array. Choosing from among five elements should reduce the running time somewhat. Implement this strategy by rewriting the quicksort code, using macros where appropriate. Write a program to test whether the running time is less. The optimal strategy for finding the pivot depends on the data in the array. Choosing from among five elements is a common strategy. What about choosing from among seven elements? If you are ambitious, try that strategy as well.

28 Suppose that `a[]` is an array of integers of size 100, and that for each `i` the element `a[i]` has value `i`. If `quicksort(a, a + 99)` is invoked, how many function calls to `quicksort()` are made? Compute this number for each version of `find_pivot()`.

29 The pointer that is returned by `partition()` is used to break the original array into two subarrays. The size of the first subarray is called the *partition break size*. Use a random-number generator to fill an array of size 100. Invoke `find_pivot()` and `partition()` to find the partition break size for the array. Do this repeatedly, say 100 times, and keep track of the running average of the break size. One expects the average break size to correspond to the middle of the array. Does this seem to be true from your experimentation?

30 The optimal break size for an array of size n is $n/2$. This identifies two subarrays of equal, or nearly equal, size for further processing. For example, given an array of size 100, a break size of 50 is optimal. Notice that a break size of 49 identifies subarrays of sizes 49 and 51, and that a break size of 51 identifies subarrays of sizes 51 and 49. Thus, the break sizes 49 and 51 are both of equal merit. Modify the program you wrote in exercise 27, on page 404, so that you keep track of the running average of the absolute value of the difference `k - 50`, where `k` is the break size obtained from `partition()`. This number corresponds inversely to how good the break size is. More generally, define

$$m = \frac{|k - (n/2)|}{n}$$

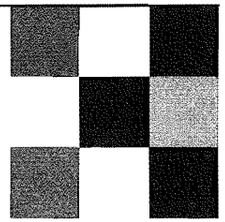
where k is the partition break size obtained from `partition()` acting on an array of size n. Fill arrays randomly and run some machine experiments to see what, if anything, can be said about m.

31 In the discussion that follows the code for the `partition()` function, we presented a table that shows the elements of the array `a[]` after each pass of the outer `while` loop. Find an array of 12 distinct elements that will produce the maximum number of passes. Write down a table that shows the elements of the array after each pass. Box the elements that were swapped. After you have done this by hand, write a program to create this table. To indicate the elements that were swapped in each pass, you can surround them with double quotes.

32 Compare `quicksort()` with `mergesort()`. (See Section 6.9, "An Example: Merge and Merge Sort," on page 266.) Time both functions with arrays having 100, 1,000, and 10,000 elements. For a small amount of data, run-time overhead dominates. That is, setting up the functions, initializing values, and other miscellaneous steps dominate the actual work required to perform the sorting. For large arrays, `quicksort()` should be faster than `mergesort()`. Is this true?

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-