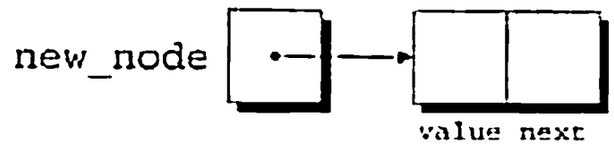
We'll use `malloc` to allocate memory for the new node, saving the return value in `new_node`:

```
new_node = malloc(sizeof(struct node));
```

`new_node` now points to a block of memory just large enough to hold a `node` structure:



Be careful to give `sizeof` the name of the *type* to be allocated, not the name of a *pointer* to that type:

```
new_node = malloc(sizeof(new_node));    /*** WRONG ***/
```

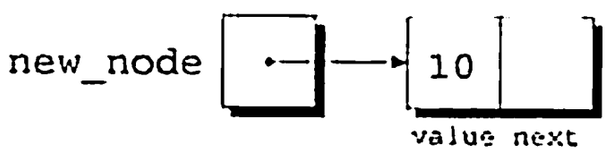
Q&A

The program will still compile, but `malloc` will allocate only enough memory for a *pointer* to a `node` structure. The likely result is a crash later, when the program attempts to store data in the node that `new_node` is presumably pointing to.

Next, we'll store data in the `value` member of the new node:

```
(*new_node).value = 10;
```

Here's how the picture will look after this assignment:



To access the `value` member of the node, we've applied the indirection operator `*` (to reference the structure to which `new_node` points), then the selection operator `.` (to select a member of the structure). The parentheses around `*new_node` are mandatory because the `.` operator would otherwise take precedence over the `*` operator.

table of operators ➤*Appendix A*

The -> Operator

Before we go on to the next step, inserting a new node into a list, let's take a moment to discuss a useful shortcut. Accessing a member of a structure using a pointer is so common that C provides a special operator just for this purpose. This operator, known as ***right arrow selection,*** is a minus sign followed by >. Using the `->` operator, we can write

```
new_node->value = 10;
```

instead of

```
(*new_node).value = 10;
```

The -> operator is a combination of the * and . operators; it performs indirection on new_node to locate the structure that it points to, then selects the value member of the structure.

lvalues ➤4.2

The -> operator produces an lvalue, so we can use it wherever an ordinary variable would be allowed. We've just seen an example in which new_node->value appears on the left side of an assignment. It could just as easily appear in a call of scanf:

```
scanf("%d", &new_node->value);
```

Notice that the & operator is still required, even though new_node is a pointer. Without the &, we'd be passing scanf the *value* of new_node->value, which has type int.

Inserting a Node at the Beginning of a Linked List

One of the advantages of a linked list is that nodes can be added at any point in the list: at the beginning, at the end, or anywhere in the middle. The beginning of a list is the easiest place to insert a node, however, so let's focus on that case.

If new_node is pointing to the node to be inserted, and first is pointing to the first node in the linked list, then we'll need two statements to insert the node into the list. First, we'll modify the new node's next member to point to the node that was previously at the beginning of the list:

```
new_node->next = first;
```

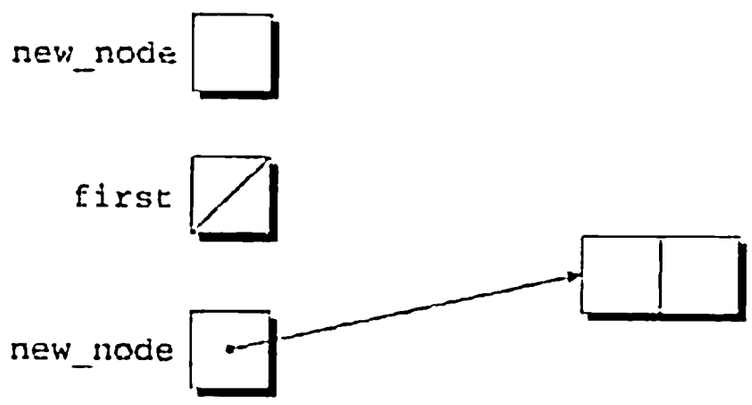
Second, we'll make first point to the new node:

```
first = new_node;
```

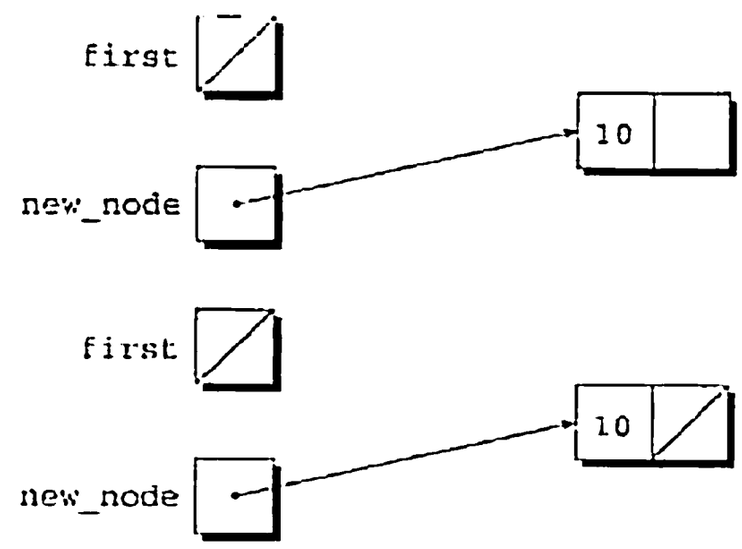
Will these statements work if the list is empty when we insert a node? Yes, fortunately. To make sure this is true, let's trace the process of inserting two nodes into an empty list. We'll insert a node containing the number 10 first, followed by a node containing 20. In the figures that follow, null pointers are shown as diagonal lines.

```
first = NULL;
```


```
new_node = malloc(sizeof(struct node));
```



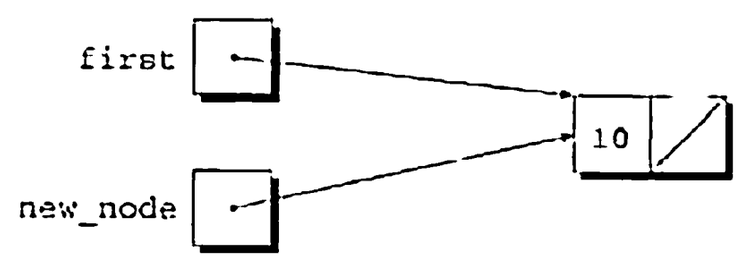
```
new_node->value = 10;
```



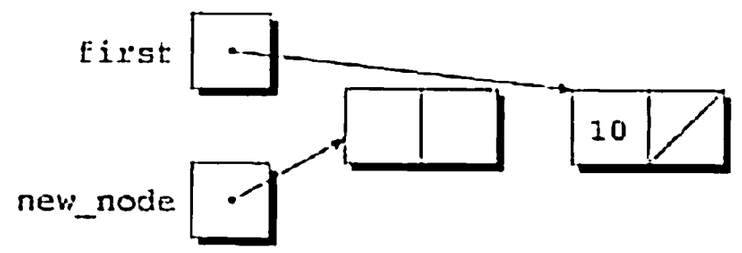

```
new_node->next = first;
```



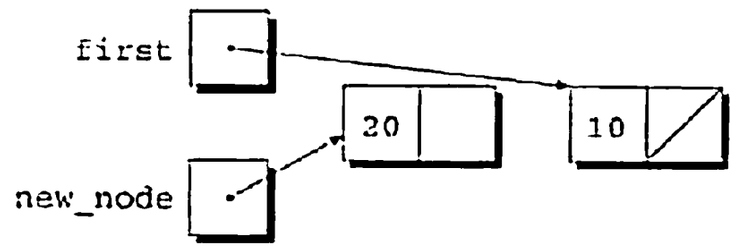
```
first = new_node;
```



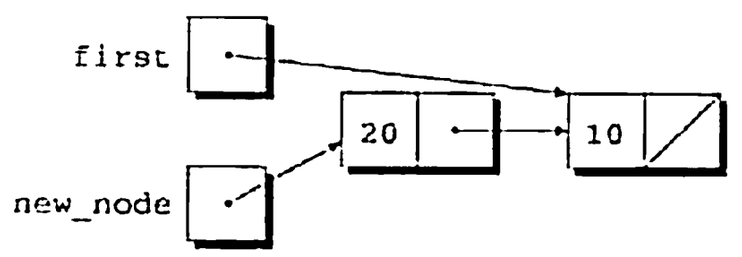

```
new_node = malloc(sizeof(struct node));
```



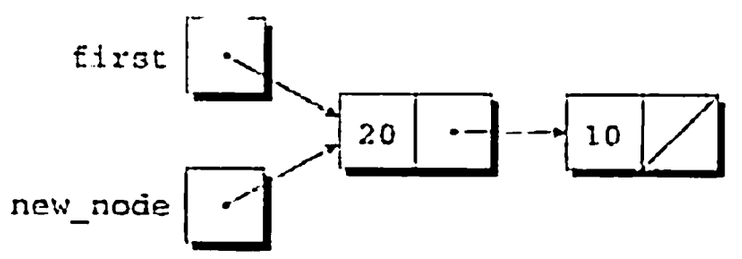

```
new_node->value = 20;
```




```
new_node->next = first;
```




```
first = new_node;
```



Inserting a node into a linked list is such a common operation that we'll probably want to write a function for that purpose. Let's name the function `add_to_list`. It will have two parameters: `list` (a pointer to the first node in the old list) and `n` (the integer to be stored in the new node).

```
struct node *add_to_list(struct node *list, int n)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if (new_node == NULL) {
    printf("Error: malloc failed in add_to_list\n");
    exit(EXIT_FAILURE);
  }
```

```
  new_node->value = n;
  new_node->next = list;
  return new_node;
}
```

Note that `add_to_list` doesn't modify the `list` pointer. Instead, it returns a pointer to the newly created node (now at the beginning of the list). When we call `add_to_list`, we'll need to store its return value into `first`:

```
first = add_to_list(first, 10);
first = add_to_list(first, 20);
```

These statements add nodes containing 10 and 20 to the list pointed to by `first`. Getting `add_to_list` to update `first` directly, rather than return a new value for `first`, turns out to be tricky. We'll return to this issue in Section 17.6.

The following function uses `add_to_list` to create a linked list containing numbers entered by the user:

```
struct node *read_numbers(void)
{
  struct node *first = NULL;
  int n;

  printf("Enter a series of integers (0 to terminate): ");
  for (;;) {
    scanf("%d", &n);
    if (n == 0)
      return first;
    first = add_to_list(first, n);
  }
}
```

The numbers will be in reverse order within the list, since `first` always points to the node containing the last number entered.

Searching a Linked List

Once we've created a linked list, we may need to search it for a particular piece of data. Although a `while` loop can be used to search a list, the `for` statement is often superior. We're accustomed to using the `for` statement when writing loops that involve counting, but its flexibility makes the `for` statement suitable for other tasks as well, including operations on linked lists. Here's the customary way to visit the nodes in a linked list, using a pointer variable `p` to keep track of the "current" node:

idiom

```
for (p = first; p != NULL; p = p->next)
  ...
```

The assignment

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

advances the p pointer from one node to the next. An assignment of this form is invariably used in C when writing a loop that traverses a linked list.

Let's write a function named `search_list` that searches a list (pointed to by the parameter `list`) for an integer n. If it finds n, `search_list` will return a pointer to the node containing n; otherwise, it will return a null pointer. Our first version of `search_list` relies on the "list-traversal" idiom:

```
struct node *search_list(struct node *list, int n)
{
  struct node *p;

  for (p = list; p != NULL; p = p->next)
    if (p->value == n)
      return p;
  return NULL;
}
```

Of course, there are many other ways to write `search_list`. One alternative would be to eliminate the p variable, instead using `list` itself to keep track of the current node:

```
struct node *search_list(struct node *list, int n)
{
  for (; list != NULL; list = list->next)
    if (list->value == n)
      return list;
  return NULL;
}
```

Since `list` is a copy of the original list pointer, there's no harm in changing it within the function.

Another alternative is to combine the `list->value == n` test with the `list != NULL` test:

```
struct node *search_list(struct node *list, int n)
{
  for (; list != NULL && list->value != n; list = list->next)
    ;
  return list;
}
```

Since `list` is NULL if we reach the end of the list, returning `list` is correct even if we don't find n. This version of `search_list` might be a bit clearer if we used a `while` statement:

```
struct node *search_list(struct node *list, int n)
{
  while (list != NULL && list->value != n)
    list = list->next;
  return list;
}
```

Deleting a Node from a Linked List

A big advantage of storing data in a linked list is that we can easily delete nodes that we no longer need. Deleting a node, like creating a node, involves three steps:

1. Locate the node to be deleted.
2. Alter the previous node so that it "bypasses" the deleted node.
3. Call `free` to reclaim the space occupied by the deleted node.

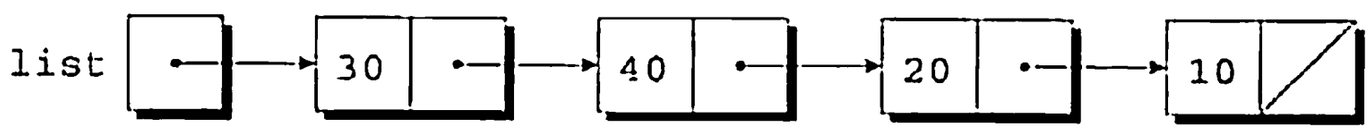
Step 1 is harder than it looks. If we search the list in the obvious way, we'll end up with a pointer to the node to be deleted. Unfortunately, we won't be able to perform step 2, which requires changing the *previous* node.

There are various solutions to this problem. We'll use the "trailing pointer" technique: as we search the list in step 1, we'll keep a pointer to the previous node (`prev`) as well as a pointer to the current node (`cur`). If `list` points to the list to be searched and `n` is the integer to be deleted, the following loop implements step 1:

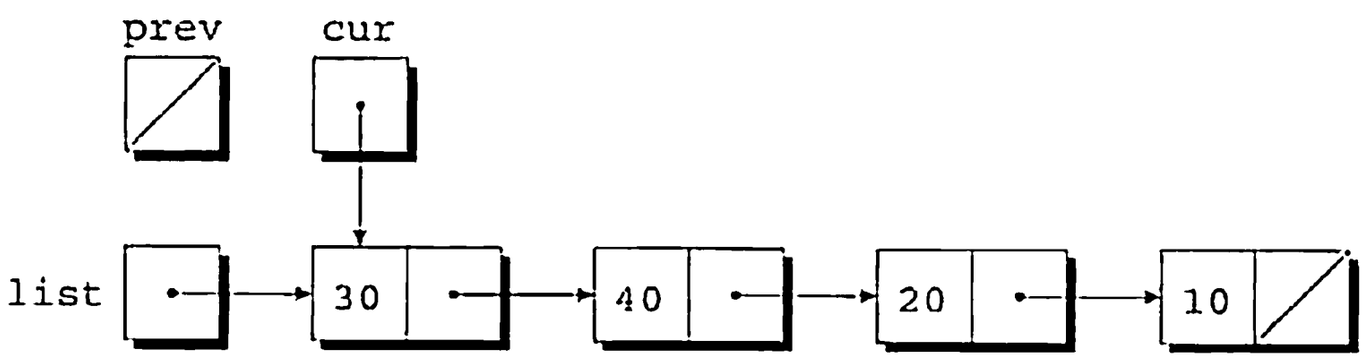
```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next)
  ;
```

Here we see the power of C's `for` statement. This rather exotic example, with its empty body and liberal use of the comma operator, performs all the actions needed to search for `n`. When the loop terminates, `cur` points to the node to be deleted, while `prev` points to the previous node (if there is one).

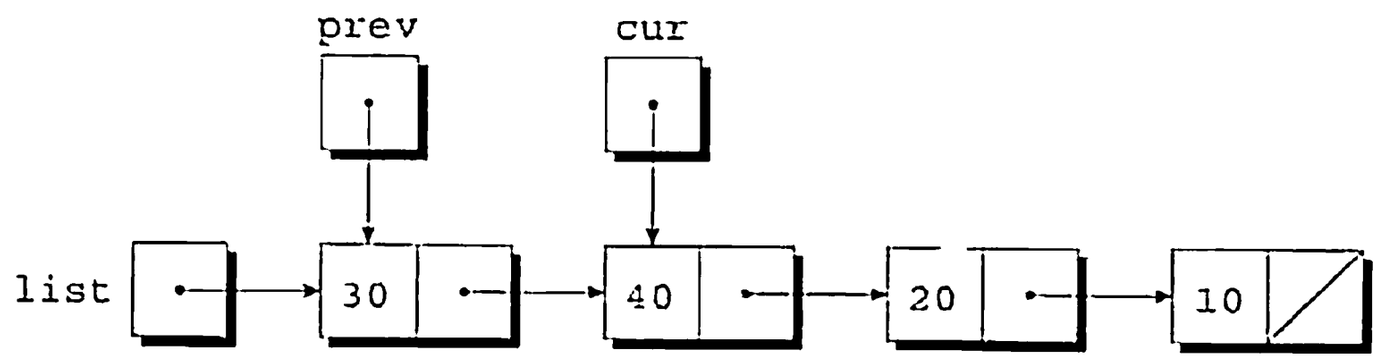
To see how this loop works, let's assume that `list` points to a list containing 30, 40, 20, and 10, in that order:



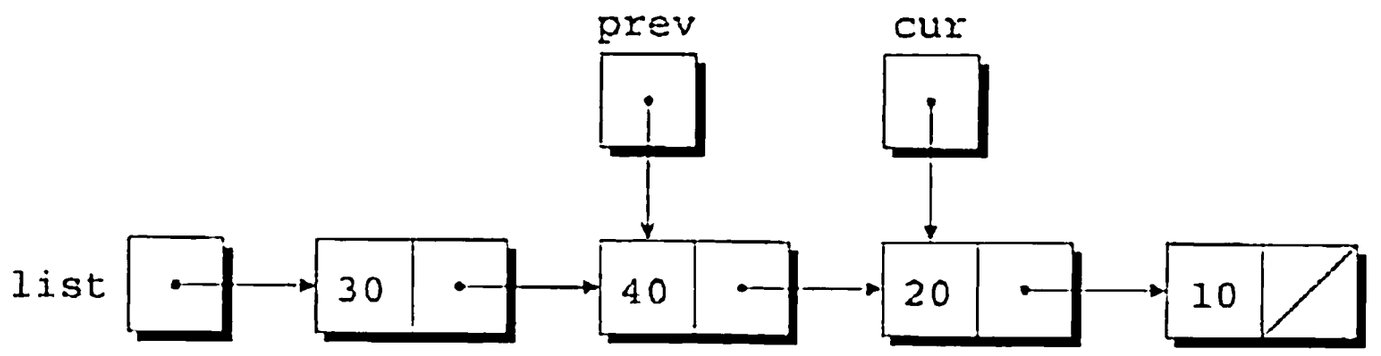
Let's say that `n` is 20, so our goal is to delete the third node in the list. After `cur = list, prev = NULL` has been executed, `cur` points to the first node in the list:



The test `cur != NULL && cur->value != n` is true, since `cur` is pointing to a node and the node doesn't contain 20. After `prev = cur, cur = cur->next` has been executed, we begin to see how the `prev` pointer will trail behind `cur`:



Again, the test `cur != NULL && cur->value != n` is true. so `prev = cur`, `cur = cur->next` is executed once more:

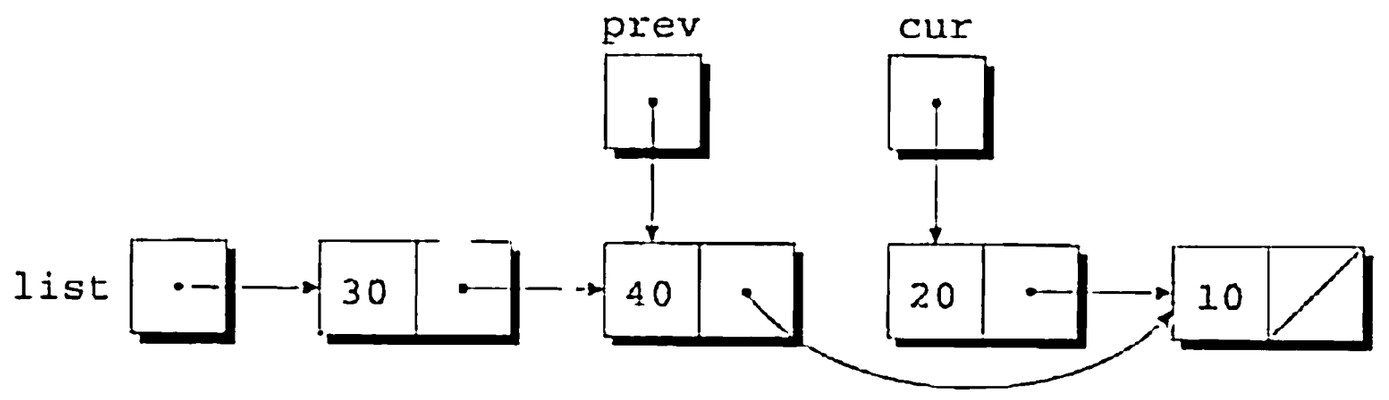


Since `cur` now points to the node containing 20, the condition `cur->value != n` is false and the loop terminates.

Next, we'll perform the bypass required by step 2. The statement

```
prev->next = cur->next;
```

makes the pointer in the previous node point to the node *after* the current node:



We're now ready for step 3, releasing the memory occupied by the current node:

```
free(cur);
```

The following function, `delete_from_list`, uses the strategy that we've just outlined. When given a list and an integer `n`, the function deletes the first node containing `n`. If no node contains `n`, `delete_from_list` does nothing. In either case, the function returns a pointer to the list.

```
struct node *delete_from_list(struct node *list, int n)
{
  struct node *cur, *prev;

  for (cur = list, prev = NULL;
       cur != NULL && cur->value != n;
       prev = cur, cur = cur->next)
    ;
```

```
  if (cur == NULL)
    return list;                    /* n was not found */
  if (prev == NULL)
    list = list->next;              /* n is in the first node */
  else
    prev->next = cur->next;         /* n is in some other node */
  free(cur);
  return list;
}
```

Deleting the first node in the list is a special case. The `prev == NULL` test checks for this case, which requires a different bypass step.

Ordered Lists

When the nodes of a list are kept in order—sorted by the data stored inside the nodes—we say that the list is ***ordered.*** Inserting a node into an ordered list is more difficult (the node won't always be put at the beginning of the list), but searching is faster (we can stop looking after reaching the point at which the desired node would have been located). The following program illustrates both the increased difficulty of inserting a node and the faster search.

PROGRAM

Maintaining a Parts Database (Revisited)

Let's redo the parts database program of Section 16.3, this time storing the database in a linked list. Using a linked list instead of an array has two major advantages: (1) We don't need to put a preset limit on the size of the database; it can grow until there's no more memory to store parts. (2) We can easily keep the database sorted by part number—when a new part is added to the database, we simply insert it in its proper place in the list. In the original program, the database wasn't sorted.

In the new program, the `part` structure will contain an additional member (a pointer to the next node in the linked list), and the variable `inventory` will be a pointer to the first node in the list:

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
  struct part *next;
};

struct part *inventory = NULL;    /* points to first part */
```

Most of the functions in the new program will closely resemble their counterparts in the original program. The `find_part` and `insert` functions will be more complex, however, since we'll keep the nodes in the `inventory` list sorted by part number.

In the original program, `find_part` returns an index into the `inventory` array. In the new program, `find_part` will return a pointer to the node that contains the desired part number. If it doesn't find the part number, `find_part` will return a null pointer. Since the `inventory` list is sorted by part number, the new version of `find_part` can save time by stopping its search when it finds a node containing a part number that's greater than or equal to the desired part number. `find_part`'s search loop will have the form

```
for (p = inventory;
     p != NULL && number > p->number;
     p = p->next)
  ;
```

The loop will terminate when `p` becomes `NULL` (indicating that the part number wasn't found) or when `number > p->number` is false (indicating that the part number we're looking for is less than or equal to a number already stored in a node). In the latter case, we still don't know whether or not the desired number is actually in the list, so we'll need another test:

```
if (p != NULL && number == p->number)
  return p;
```

The original version of `insert` stores a new part in the next available array element. The new version must determine where the new part belongs in the list and insert it there. We'll also have `insert` check whether the part number is already present in the list. `insert` can accomplish both tasks by using a loop similar to the one in `find_part`:

```
for (cur = inventory, prev = NULL;
     cur != NULL && new_node->number > cur->number;
     prev = cur, cur = cur->next)
  ;
```

This loop relies on two pointers: `cur`, which points to the current node, and `prev`, which points to the previous node. Once the loop terminates, `insert` will check whether `cur` isn't `NULL` and `new_node->number` equals `cur->number`; if so, the part number is already in the list. Otherwise `insert` will insert a new node between the nodes pointed to by `prev` and `cur`, using a strategy similar to the one we employed for deleting a node. (This strategy works even if the new part number is larger than any in the list; in that case, `cur` will be `NULL` but `prev` will point to the last node in the list.)

Here's the new program. Like the original program, this version requires the `read_line` function described in Section 16.3; I assume that `readline.h` contains a prototype for this function.

inventory2.c

```
/* Maintains a parts database (linked list version) */

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

```
#define NAME_LEN 25

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
  struct part *next;
};

struct part *inventory = NULL;   /* points to first part */

struct part *find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);

/**********************************************************
 * main: Prompts the user to enter an operation code,     *
 *       then calls a function to perform the requested   *
 *       action. Repeats until the user enters the        *
 *       command 'q'. Prints an error message if the user *
 *       enters an illegal code.                          *
 **********************************************************/
int main(void)
{
  char code;

  for (;;) {
    printf("Enter operation code: ");
    scanf(" %c", &code);
    while (getchar() != '\n')   /* skips to end of line */
      ;
    switch (code) {
      case 'i': insert();
                break;
      case 's': search();
                break;
      case 'u': update();
                break;
      case 'p': print();
                break;
      case 'q': return 0;
      default:  printf("Illegal code\n");
    }
    printf("\n");
  }
}

/**********************************************************
 * find_part: Looks up a part number in the inventory     *
 *            list. Returns a pointer to the node         *
 *            containing the part number; if the part     *
 *            number is not found, returns NULL.          *
 **********************************************************/
```

```
struct part *find_part(int number)
{
  struct part *p;

  for (p = inventory;
       p != NULL && number > p->number;
       p = p->next)
    ;
  if (p != NULL && number == p->number)
    return p;
  return NULL;
}

/**********************************************************
 * insert: Prompts the user for information about a new   *
 *         part and then inserts the part into the        *
 *         inventory list; the list remains sorted by     *
 *         part number. Prints an error message and       *
 *         returns prematurely if the part already exists *
 *         or space could not be allocated for the part.  *
 **********************************************************/
void insert(void)
{
  struct part *cur, *prev, *new_node;

  new_node = malloc(sizeof(struct part));
  if (new_node == NULL) {
    printf("Database is full; can't add more parts.\n");
    return;
  }

  printf("Enter part number: ");
  scanf("%d", &new_node->number);

  for (cur = inventory, prev = NULL;
       cur != NULL && new_node->number > cur->number;
       prev = cur, cur = cur->next)
    ;
  if (cur != NULL && new_node->number == cur->number) {
    printf("Part already exists.\n");
    free(new_node);
    return;
  }

  printf("Enter part name: ");
  read_line(new_node->name, NAME_LEN);
  printf("Enter quantity on hand: ");
  scanf("%d", &new_node->on_hand);

  new_node->next = cur;
  if (prev == NULL)
    inventory = new_node;
  else
    prev->next = new_node;
}
```

```
/**********************************************
 * search: Prompts the user to enter a part number, then  *
 *         looks up the part in the database. If the part *
 *         exists, prints the name and quantity on hand;  *
 *         if not, prints an error message.               *
 **********************************************/
void search(void)
{
  int number;
  struct part *p;

  printf("Enter part number: ");
  scanf("%d", &number);
  p = find_part(number);
  if (p != NULL) {
    printf("Part name: %s\n", p->name);
    printf("Quantity on hand: %d\n", p->on_hand);
  } else
    printf("Part not found.\n");
}

/**********************************************
 * update: Prompts the user to enter a part number.       *
 *         Prints an error message if the part doesn't    *
 *         exist; otherwise, prompts the user to enter    *
 *         change in quantity on hand and updates the     *
 *         database.                                      *
 **********************************************/
void update(void)
{
  int number, change;
  struct part *p;

  printf("Enter part number: ");
  scanf("%d", &number);
  p = find_part(number);
  if (p != NULL) {
    printf("Enter change in quantity on hand: ");
    scanf("%d", &change);
    p->on_hand += change;
  } else
    printf("Part not found.\n");
}

/**********************************************
 * print: Prints a listing of all parts in the database,  *
 *        showing the part number, part name, and         *
 *        quantity on hand. Part numbers will appear in   *
 *        ascending order.                                *
 **********************************************/
void print(void)
{
  struct part *p;
```

```
  printf("Part Number   Part Name                      "
         "Quantity on Hand\n");
  for (p = inventory; p != NULL; p = p->next)
    printf("%7d       %-25s%11d\n", p->number, p->name,
           p->on_hand);
}
```

Notice the use of `free` in the `insert` function. `insert` allocates memory for a part before checking to see if the part already exists. If it does, `insert` releases the space to avoid a memory leak.

17.6 Pointers to Pointers

In Section 13.7, we came across the notion of a *pointer* to a *pointer*. In that section, we used an array whose elements were of type `char *`; a pointer to one of the array elements itself had type `char **`. The concept of "pointers to pointers" also pops up frequently in the context of linked data structures. In particular, when an argument to a function is a pointer variable, we'll sometimes want the function to be able to modify the variable by making it point somewhere else. Doing so requires the use of a pointer to a pointer.

Consider the `add_to_list` function of Section 17.5, which inserts a node at the beginning of a linked list. When we call `add_to_list`, we pass it a pointer to the first node in the original list; it then returns a pointer to the first node in the updated list:

```
struct node *add_to_list(struct node *list, int n)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if (new_node == NULL) {
    printf("Error: malloc failed in add_to_list\n");
    exit(EXIT_FAILURE);
  }
  new_node->value = n;
  new_node->next = list;
  return new_node;
}
```

Suppose that we modify the function so that it assigns `new_node` to `list` instead of returning `new_node`. In other words, let's remove the `return` statement from `add_to_list` and replace it by

```
list = new_node;
```

Unfortunately, this idea doesn't work. Suppose that we call `add_to_list` in the following way:

```
add_to_list(first, 10);
```

At the point of the call, `first` is copied into `list`. (Pointers, like all arguments, are passed by value.) The last line in the function changes the value of `list`, making it point to the new node. This assignment doesn't affect `first`, however.

Getting `add_to_list` to modify `first` is possible, but it requires passing `add_to_list` a *pointer* to `first`. Here's the correct version of the function:

```
void add_to_list(struct node **list, int n)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if (new_node == NULL) {
    printf("Error: malloc failed in add_to_list\n");
    exit(EXIT_FAILURE);
  }
  new_node->value = n;
  new_node->next = *list;
  *list = new_node;
}
```

When we call the new version of `add_to_list`, the first argument will be the address of `first`:

```
add_to_list(&first, 10);
```

Since `list` is assigned the address of `first`, we can use `*list` as an alias for `first`. In particular, assigning `new_node` to `*list` will modify `first`.

17.7 Pointers to Functions

We've seen that pointers may point to various kinds of data, including variables, array elements, and dynamically allocated blocks of memory. But C doesn't require that pointers point only to *data;* it's also possible to have pointers to *functions.* Pointers to functions aren't as odd as you might think. After all, functions occupy memory locations, so every function has an address, just as each variable has an address.

Function Pointers as Arguments

We can use function pointers in much the same way we use pointers to data. In particular, passing a function pointer as an argument is fairly common in C. Suppose that we're writing a function named `integrate` that integrates a mathematical function `f` between points `a` and `b`. We'd like to make `integrate` as general as possible by passing it `f` as an argument. To achieve this effect in C, we'll declare `f` to be a pointer to a function. Assuming that we want to integrate functions that have

a `double` parameter and return a `double` result, the prototype for `integrate` will look like this:

```
double integrate(double (*f)(double), double a, double b);
```

The parentheses around `*f` indicate that `f` is a pointer to a function, not a function that returns a pointer. It's also legal to declare `f` as though it were a function:

```
double integrate(double f(double), double a, double b);
```

From the compiler's standpoint, this prototype is identical to the previous one.

When we call `integrate`, we'll supply a function name as the first argument. For example, the following call will integrate the `sin` (sine) function from 0 to $\pi/2$:

sin function ➤23.3

```
result = integrate(sin, 0.0, PI / 2);
```

Notice that there are no parentheses after `sin`. When a function name isn't followed by parentheses, the C compiler produces a pointer to the function instead of generating code for a function call. In our example, we're not calling `sin`; instead, we're passing `integrate` a pointer to `sin`. If this seems confusing, think of how C handles arrays. If `a` is the name of an array, then `a[i]` represents one element of the array, while `a` by itself serves as a pointer to the array. In a similar way, if `f` is a function, C treats `f(x)` as a *call* of the function but `f` by itself as a *pointer* to the function.

Within the body of `integrate`, we can call the function that `f` points to:

```
y = (*f)(x);
```

`*f` represents the function that `f` points to; `x` is the argument to the call. Thus, during the execution of `integrate(sin, 0.0, PI / 2)`, each call of `*f` is actually a call of `sin`. As an alternative to `(*f)(x)`, C allows us to write `f(x)` to call the function that `f` points to. Although `f(x)` looks more natural, I'll stick with `(*f)(x)` as a reminder that `f` is a pointer to a function, not a function name.

The `qsort` Function

Although it might seem that pointers to functions aren't relevant to the average programmer, that couldn't be further from the truth. In fact, some of the most useful functions in the C library require a function pointer as an argument. One of these is `qsort`, which belongs to the `<stdlib.h>` header. `qsort` is a general-purpose sorting function that's capable of sorting any array, based on any criteria that we choose.

Q&A

Since the elements of the array that it sorts may be of any type—even a structure or union type—`qsort` must be told how to determine which of two array elements is "smaller." We'll provide this information to `qsort` by writing a ***comparison function.*** When given two pointers `p` and `q` to array elements, the comparison function must return an integer that is *negative* if `*p` is "less than" `*q`,

zero if *p is "equal to" *q, and *positive* if *p is "greater than" *q. The terms "less than," "equal to," and "greater than" are in quotes because it's our responsibility to determine how *p and *q are compared.

qsort has the following prototype:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

base must point to the first element in the array. (If only a portion of the array is to be sorted, we'll make base point to the first element in this portion.) In the simplest case, base is just the name of the array. nmemb is the number of elements to be sorted (not necessarily the number of elements in the array). size is the size of each array element, measured in bytes. compar is a pointer to the comparison function. When qsort is called, it sorts the array into ascending order, calling the comparison function whenever it needs to compare array elements.

To sort the inventory array of Section 16.3, we'd use the following call of qsort:

Q&A

```
qsort(inventory, num_parts, sizeof(struct part), compare_parts);
```

Notice that the second argument is num_parts, not MAX_PARTS; we don't want to sort the entire inventory array, just the portion in which parts are currently stored. The last argument, compare_parts, is a function that compares two part structures.

Writing the compare_parts function isn't as easy as you might expect. qsort requires that its parameters have type void *, but we can't access the members of a part structure through a void * pointer; we need a pointer of type struct part * instead. To solve the problem, we'll have compare_parts assign its parameters, p and q, to variables of type struct part *, thereby converting them to the desired type. compare_parts can now use these variables to access the members of the structures that p and q point to. Assuming that we want to sort the inventory array into ascending order by part number, here's how the compare_parts function might look:

```
int compare_parts(const void *p, const void *q)
{
  const struct part *p1 = p;
  const struct part *q1 = q;

  if (p1->number < q1->number)
    return -1;
  else if (p1->number == q1->number)
    return 0;
  else
    return 1;
}
```

The declarations of p1 and q1 include the word const to avoid getting a warning from the compiler. Since p and q are const pointers (indicating that the objects

to which they point should not be modified), they should be assigned only to pointer variables that are also declared to be `const`.

Although this version of `compare_parts` works, most C programmers would write the function more concisely. First, notice that we can replace `p1` and `q1` by cast expressions:

```
int compare_parts(const void *p, const void *q)
{
  if (((struct part *) p)->number <
      ((struct part *) q)->number)
    return -1;
  else if (((struct part *) p)->number ==
           ((struct part *) q)->number)
    return 0;
  else
    return 1;
}
```

The parentheses around `((struct part *) p)` are necessary; without them, the compiler would try to cast `p->number` to type `struct part *`.

We can make `compare_parts` even shorter by removing the `if` statements:

```
int compare_parts(const void *p, const void *q)
{
  return ((struct part *) p)->number -
         ((struct part *) q)->number;
}
```

Subtracting q's part number from p's part number produces a negative result if p has a smaller part number, zero if the part numbers are equal, and a positive result if p has a larger part number. (Note that subtracting two integers is potentially risky because of the danger of overflow. I'm assuming that part numbers are positive integers, so that shouldn't happen here.)

To sort the `inventory` array by part name instead of part number, we'd use the following version of `compare_parts`:

```
int compare_parts(const void *p, const void *q)
{
  return strcmp(((struct part *) p)->name,
                ((struct part *) q)->name);
}
```

All `compare_parts` has to do is call `strcmp`, which conveniently returns a negative, zero, or positive result.

Other Uses of Function Pointers

Although I've emphasized the usefulness of function pointers as arguments to other functions, that's not all they're good for. C treats pointers to functions just like pointers to data: we can store function pointers in variables or use them as ele-

ments of an array or as members of a structure or union. We can even write functions that return function pointers.

Here's an example of a variable that can store a pointer to a function:

```
void (*pf)(int);
```

`pf` can point to any function with an `int` parameter and a return type of `void`. If `f` is such a function, we can make `pf` point to `f` in the following way:

```
pf = f;
```

Notice that there's no ampersand preceding `f`. Once `pf` points to `f`, we can call `f` by writing either

```
(*pf)(i);
```

or

```
pf(i);
```

Arrays whose elements are function pointers have a surprising number of applications. For example, suppose that we're writing a program that displays a menu of commands for the user to choose from. We can write functions that implement these commands, then store pointers to the functions in an array:

```
void (*file_cmd[])(void) = {new_cmd,
                            open_cmd,
                            close_cmd,
                            close_all_cmd,
                            save_cmd,
                            save_as_cmd,
                            save_all_cmd,
                            print_cmd,
                            exit_cmd
                           };
```

If the user selects command n, where n falls between 0 and 8, we can subscript the `file_cmd` array and call the corresponding function:

```
(*file_cmd[n])();    /* or file_cmd[n](); */
```

Of course, we could get a similar effect with a `switch` statement. Using an array of function pointers gives us more flexibility, however, since the elements of the array can be changed as the program is running.

PROGRAM Tabulating the Trigonometric Functions

<math.h> header ➤23.3

The following program prints tables showing the values of the `cos`, `sin`, and `tan` functions (all three belong to `<math.h>`). The program is built around a function named `tabulate` that, when passed a function pointer `f`, prints a table showing the values of `f`.

tabulate.c

```
/* Tabulates values of trigonometric functions */

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

void tabulate(double (*f)(double), double first,
              double last, double incr);

int main(void)
{
  double final, increment, initial;

  printf("Enter initial value: ");
  scanf("%lf", &initial);

  printf("Enter final value: ");
  scanf("%lf", &final);

  printf("Enter increment: ");
  scanf("%lf", &increment);

  printf("\n      x         cos(x)"
         "\n   -------    -------\n");
  tabulate(cos, initial, final, increment);

  printf("\n      x         sin(x)"
         "\n   -------    -------\n");
  tabulate(sin, initial, final, increment);

  printf("\n      x         tan(x)"
         "\n   -------    -------\n");
  tabulate(tan, initial, final, increment);

  return 0;
}

void tabulate(double (*f)(double), double first,
              double last, double incr)
{
  double x;
  int i, num_intervals;

  num_intervals = ceil((last - first) / incr);
  for (i = 0; i <= num_intervals; i++) {
    x = first + i * incr;
    printf("%10.5f %10.5f\n", x, (*f)(x));
  }
}
```

`tabulate` uses the `ceil` function, which also in `<math.h>`. When given an argument `x` of `double` type, `ceil` returns the smallest integer that's greater than or equal to `x`.

Here's what a session with `tabulate.c` might look like:

```
Enter initial value: 0
Enter final value: .5
Enter increment: .1

      x          cos(x)
-------     -------
0.00000     1.00000
0.10000     0.99500
0.20000     0.98007
0.30000     0.95534
0.40000     0.92106
0.50000     0.87758

      x          sin(x)
-------     -------
0.00000     0.00000
0.10000     0.09983
0.20000     0.19867
0.30000     0.29552
0.40000     0.38942
0.50000     0.47943

      x          tan(x)
-------     -------
0.00000     0.00000
0.10000     0.10033
0.20000     0.20271
0.30000     0.30934
0.40000     0.42279
0.50000     0.54630
```

17.8 Restricted Pointers (C99)

This section and the next discuss two of C99's pointer-related features. Both are primarily of interest to advanced C programmers; most readers will want to skip these sections.

In C99, the keyword `restrict` may appear in the declaration of a pointer:

```
int * restrict p;
```

A pointer that's been declared using `restrict` is called a ***restricted pointer.*** The intent is that if `p` points to an object that is later modified, then that object is not accessed in any way other than through `p`. (Alternative ways to access the object include having another pointer to the same object or having `p` point to a named variable.) Having more than one way to access an object is often called ***aliasing.***

Let's look at an example of the kind of behavior that restricted pointers are supposed to discourage. Suppose that `p` and `q` have been declared as follows:

```
int * restrict p;
int * restrict q;
```

Now suppose that p is made to point to a dynamically allocated block of memory:

```
p = malloc(sizeof(int));
```

(A similar situation would arise if p were assigned the address of a variable or an array element.) Normally it would be legal to copy p into q and then modify the integer through q:

```
q = p;
*q = 0;    /* causes undefined behavior */
```

Because p is a restricted pointer, however, the effect of executing the statement `*q = 0;` is undefined. By making p and q point to the same object, we caused `*p` and `*q` to be aliases.

extern storage class ➤ 18.2

If a restricted pointer p is declared as a local variable without the `extern` storage class, `restrict` applies only to p when the block in which p is declared is being executed. (Note that the body of a function is a block.) `restrict` can be used with function parameters of pointer type, in which case it applies only when the function is executing. When `restrict` is applied to a pointer variable with file scope, however, the restriction lasts for the entire execution of the program.

blocks ➤ 10.3

file scope ➤ 10.2

The exact rules for using `restrict` are rather complex; see the C99 standard for details. There are even situations in which an alias created from a restricted pointer is legal. For example, a restricted pointer p can be legally copied into another restricted pointer variable q, provided that p is local to a function and q is defined inside a block nested within the function's body.

To illustrate the use of `restrict`, let's look at the memcpy and memmove functions, which belong to the `<string.h>` header. memcpy has the following prototype in C99:

<string.h> header ➤ 23.6

```
void *memcpy(void * restrict s1, const void * restrict s2,
             size_t n);
```

memcpy is similar to `strcpy`, except that it copies bytes from one object to another (`strcpy` copies characters from one string into another). `s2` points to the data to be copied, `s1` points to the destination of the copy, and `n` is the number of bytes to be copied. The use of `restrict` with both `s1` and `s2` indicates that the source of the copy and the destination shouldn't overlap. (It doesn't *guarantee* that they don't overlap, however.)

In contrast, `restrict` doesn't appear in the prototype for memmove:

```
void *memmove(void *s1, const void *s2, size_t n);
```

memmove does the same thing as memcpy: it copies bytes from one place to another. The difference is that memmove is guaranteed to work even if the source and destination overlap. For example, we could use memmove to shift the elements of an array by one position:

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

```
memmove(&a[0], &a[1], 99 * sizeof(int));
```

Prior to C99, there was no way to document the difference between memcpy and memmove. The prototypes for the two functions were nearly identical:

```
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
```

The use of `restrict` in the C99 version of memcpy's prototype lets the programmer know that `s1` and `s2` should point to objects that don't overlap, or else the function isn't guaranteed to work.

Although using `restrict` in function prototypes is useful documentation, that's not the primary reason for its existence. `restrict` provides information to the compiler that may enable it to produce more efficient code—a process known as ***optimization***. (The `register` storage class serves the same purpose.) Not every compiler attempts to optimize programs, however, and the ones that do normally allow the programmer to disable optimization. As a result, the C99 standard guarantees that `restrict` has no effect on the behavior of a program that conforms to the standard: if all uses of `restrict` are removed from such a program, it should behave the same.

`register` storage class ➤ 18.2

Most programmers won't use `restrict` unless they're fine-tuning a program to achieve the best possible performance. Still, it's worth knowing about `restrict` because it appears in the C99 prototypes for a number of standard library functions.

17.9 Flexible Array Members (C99)

Every once in a while, we'll need to define a structure that contains an array of an unknown size. For example, we might want to store strings in a form that's different from the usual one. Normally, a string is an array of characters, with a null character marking the end. However, there are advantages to storing strings in other ways. One alternative is to store the length of the string along with the string's characters (but with no null character). The length and the characters could be stored in a structure such as this one:

```
struct vstring {
  int len;
  char chars[N];
};
```

Here `N` is a macro that represents the maximum length of a string. Using a fixed-length array such as this is undesirable, however, because it forces us to limit the length of the string, plus it wastes memory (since most strings won't need all `N` characters in the array).

C programmers have traditionally solved this problem by declaring the length of `chars` to be 1 (a dummy value) and then dynamically allocating each string:

```
struct vstring {
  int len;
  char chars[1];
};
...
struct vstring *str = malloc(sizeof(struct vstring) + n - 1);
str->len = n;
```

We're "cheating" by allocating more memory than the structure is declared to have (in this case, an extra n – 1 characters), and then using the memory to store additional elements of the `chars` array. This technique has become so common over the years that it has a name: the "struct hack."

The struct hack isn't limited to character arrays: it has a variety of uses. Over time, it has become popular enough to be supported by many compilers. Some (including GCC) even allow the `chars` array to have zero length, which makes this trick a little more explicit. Unfortunately, the C89 standard doesn't guarantee that the struct hack will work, nor does it allow zero-length arrays.

In recognition of the struct hack's usefulness, C99 has a feature known as the ***flexible array member*** that serves the same purpose. When the last member of a structure is an array, its length may be omitted:

```
struct vstring {
  int len;
  char chars[];   /* flexible array member - C99 only */
};
```

The length of the `chars` array isn't determined until memory is allocated for a `vstring` structure, normally using a call of `malloc`:

```
struct vstring *str = malloc(sizeof(struct vstring) + n);
str->len = n;
```

In this example, `str` points to a `vstring` structure in which the `chars` array occupies `n` characters. The `sizeof` operator ignores the `chars` member when computing the size of the structure. (A flexible array member is unusual in that it takes up no space within a structure.)

A few special rules apply to a structure that contains a flexible array member. The flexible array member must appear last in the structure, and the structure must have at least one other member. Copying a structure that contains a flexible array member will copy the other members but not the flexible array itself.

A structure that contains a flexible array member is an ***incomplete type***. An incomplete type is missing part of the information needed to determine how much memory it requires. Incomplete types, which are discussed further in one of the Q&A questions at the end of this chapter and in Section 19.3, are subject to various restrictions. In particular, an incomplete type (and hence a structure that contains a flexible array member) can't be a member of another structure or an element of an array. However, an array may contain pointers to structures that have a flexible array member; Programming Project 7 at the end of this chapter is built around such an array.

Q & A

Q: What does the NULL macro represent? [p. 415]

A: NULL actually stands for 0. When we use 0 in a context where a pointer would be required, C compilers treat it as a null pointer instead of the integer 0. The NULL macro is provided merely to help avoid confusion. The assignment

```
p = 0;
```

could be assigning the value 0 to a numeric variable or assigning a null pointer to a pointer variable; we can't easily tell which. In contrast, the assignment

```
p = NULL;
```

makes it clear that p is a pointer.

***Q: In the header files that come with my compiler, NULL is defined as follows:**

```
#define NULL (void *) 0
```

What's the advantage of casting 0 to `void *`?

A: This trick, which is allowed by the C standard, enables compilers to spot incorrect uses of the null pointer. For example, suppose that we try to assign NULL to an integer variable:

```
i = NULL;
```

If NULL is defined as 0, this assignment is perfectly legal. But if NULL is defined as `(void *) 0`, the compiler can warn us that we're assigning a pointer to an integer variable.

variable-length argument lists ➤*26.1*

Defining NULL as `(void *) 0` has a second, more important, advantage. Suppose that we call a function with a variable-length argument list and pass NULL as one of the arguments. If NULL is defined as 0, the compiler will incorrectly pass a zero integer value. (In an ordinary function call, NULL works fine because the compiler knows from the function's prototype that it expects a pointer. When a function has a variable-length argument list, however, the compiler lacks this knowledge.) If NULL is defined as `(void *) 0`, the compiler will pass a null pointer.

To make matters even more confusing, some header files define NULL to be `0L` (the `long` version of 0). This definition, like the definition of NULL as 0, is a holdover from C's earlier years, when pointers and integers were compatible. For most purposes, though, it really doesn't matter how NULL is defined; just think of it as a name for the null pointer.

Q: Since 0 is used to represent the null pointer, I guess a null pointer is just an address with all zero bits, right?

A: Not necessarily. Each C compiler is allowed to represent null pointers in a different way, and not all compilers use a zero address. For example, some compilers use a nonexistent memory address for the null pointer; that way, attempting to access memory through a null pointer can be detected by the hardware.

How the null pointer is stored inside the computer shouldn't concern us; that's a detail for compiler experts to worry about. The important thing is that, when used in a pointer context, `0` is converted to the proper internal form by the compiler.

Q: Is it acceptable to use `NULL` as a null character?

A: Definitely not. `NULL` is a macro that represents the null *pointer*, not the null *character*. Using `NULL` as a null character will work with some compilers, but not with all (since some define `NULL` as `(void *) 0`). In any event, using `NULL` as anything other than a pointer can lead to a great deal of confusion. If you want a name for the null character, define the following macro:

```
#define NUL '\0'
```

***Q: When my program terminates, I get the message *"Null pointer assignment."* What does this mean?**

A: This message, which is produced by programs compiled with some older DOS-based C compilers, indicates that the program has stored data in memory using a bad pointer (but not necessarily a null pointer). Unfortunately, the message isn't displayed until the program terminates, so there's no clue as to which statement caused the error. The *"Null pointer assignment"* message can be caused by a missing `&` in `scanf`:

```
scanf("%d", i);    /* should have been scanf("%d", &i); */
```

Another possibility is an assignment involving a pointer that's uninitialized or null:

```
*p = i;    /* p is uninitialized or null */
```

***Q: How does a program know that a "null pointer assignment" has occurred?**

A: The message depends on the fact that, in the small and medium memory models, data is stored in a single segment, with addresses beginning at 0. The compiler leaves a "hole" at the beginning of the data segment—a small block of memory that's initialized to 0 but otherwise isn't used by the program. When the program terminates, it checks to see if any data in the "hole" area is nonzero. If so, it must have been altered through a bad pointer.

Q: Is there any advantage to casting the return value of `malloc` or the other memory allocation functions? [p. 416]

A: Not usually. Casting the `void *` pointer that these functions return is unnecessary, since pointers of type `void *` are automatically converted to any pointer type upon assignment. The habit of casting the return value is a holdover from older versions of C, in which the memory allocation functions returned a `char *` value, making the cast necessary. Programs that are designed to be compiled as C++ code

may benefit from the cast, but that's about the only reason to do it.

In C89, there's actually a small advantage to *not* performing the cast. Suppose that we've forgotten to include the `<stdlib.h>` header in our program. When we call `malloc`, the compiler will assume that its return type is `int` (the default return value for any C function). If we don't cast the return value of `malloc`, a C89 compiler will produce an error (or at least a warning), since we're trying to assign an integer value to a pointer variable. On the other hand, if we cast the return value to a pointer, the program may compile, but likely won't run properly. With C99, this advantage disappears. Forgetting to include the `<stdlib.h>` header will cause an error when `malloc` is called, because C99 requires that a function be declared before it's called.

Q: The `calloc` function initializes a memory block by setting its bits to zero. Does this mean that all data items in the block become zero? [p. 421]

A: Usually, but not always. Setting an integer to zero bits always makes the integer zero. Setting a floating-point number to zero bits usually makes the number zero, but this isn't guaranteed—it depends on how floating-point numbers are stored. The story is the same for pointers; a pointer whose bits are zero isn't necessary a null pointer.

***Q: I see how the structure tag mechanism allows a structure to contain a pointer to itself. But what if two structures each have a member that points to the other? [p. 425]**

A: Here's how we'd handle that situation:

```
struct s1;    /* incomplete declaration of s1 */

struct s2 {
  …
  struct s1 *p;
  …
};

struct s1 {
  …
  struct s2 *q;
  …
};
```

incomplete types ➤ 19.3

The first declaration of `s1` creates an incomplete structure type, since we haven't specified the members of `s1`. The second declaration of `s1` "completes" the type by describing the members of the structure. Incomplete declarations of a structure type are permitted in C, although their uses are limited. Creating a pointer to such a type (as we did when declaring `p`) is one of these uses.

Q: Calling `malloc` with the wrong argument—causing it to allocate too much memory or too little memory—seems to be a common error. Is there a safer way to use `malloc`? [p. 426]

A: Yes, there is. Some programmers use the following idiom when calling `malloc` to allocate memory for a single object:

```
p = malloc(sizeof(*p));
```

Since `sizeof(*p)` is the size of the object to which `p` will point, this statement guarantees that the correct amount of memory will be allocated. At first glance, this idiom looks fishy: it's likely that `p` is uninitialized, making the value of `*p` undefined. However, `sizeof` doesn't evaluate `*p`, it merely computes its size, so the idiom works even if `p` is uninitialized or contains a null pointer.

To allocate memory for an array with `n` elements, we can use a slightly modified version of the idiom:

```
p = malloc(n * sizeof(*p));
```

Q: Why isn't the `qsort` function simply named `sort`? [p. 440]

A: The name `qsort` comes from the Quicksort algorithm published by C. A. R. Hoare in 1962 (and discussed in Section 9.6). Ironically, the C standard doesn't require that `qsort` use the Quicksort algorithm, although many versions of `qsort` do.

Q: Isn't it necessary to cast `qsort`'s first argument to type `void *`, as in the following example? [p. 441]

```
qsort((void *) inventory, num_parts, sizeof(struct part),
      compare_parts);
```

A: No. A pointer of any type can be converted to `void *` automatically.

***Q: I want to use `qsort` to sort an array of integers, but I'm having trouble writing a comparison function. What's the secret?**

A: Here's a version that works:

```
int compare_ints(const void *p, const void *q)
{
  return *(int *)p - *(int *)q;
}
```

Bizarre, eh? The expression `(int *)p` casts `p` to type `int *`, so `*(int *)p` would be the integer that `p` points to. A word of warning, though: Subtracting two integers may cause overflow. If the integers being sorted are completely arbitrary, it's safer to use `if` statements to compare `*(int *)p` with `*(int *)q`.

***Q: I needed to sort an array of strings, so I figured I'd just use `strcmp` as the comparison function. When I passed it to `qsort`, however, the compiler gave me a warning. I tried to fix the problem by embedding `strcmp` in a comparison function:**

```
int compare_strings(const void *p, const void *q)
{
  return strcmp(p, q);
}
```

Now my program compiles, but `qsort` doesn't seem to sort the array. What am I doing wrong?

A: First, you can't pass `strcmp` itself to `qsort`, since `qsort` requires a comparison function with two `const void *` parameters. Your `compare_strings` function doesn't work because it incorrectly assumes that `p` and `q` are strings (`char *` pointers). In fact, `p` and `q` point to array elements containing `char *` pointers. To fix `compare_strings`, we'll cast `p` and `q` to type `char **`, then use the `*` operator to remove one level of indirection:

```
int compare_strings(const void *p, const void *q)
{
  return strcmp(*(char **)p, *(char **)q);
}
```

Exercises

Section 17.1

1. Having to check the return value of `malloc` (or any other memory allocation function) each time we call it can be an annoyance. Write a function named `my_malloc` that serves as a "wrapper" for `malloc`. When we call `my_malloc` and ask it to allocate `n` bytes, it in turn calls `malloc`, tests to make sure that `malloc` doesn't return a null pointer, and then returns the pointer from `malloc`. Have `my_malloc` print an error message and terminate the program if `malloc` returns a null pointer.

Section 17.2

W 2. Write a function named `duplicate` that uses dynamic storage allocation to create a copy of a string. For example, the call

```
p = duplicate(str);
```

would allocate space for a string of the same length as `str`, copy the contents of `str` into the new string, and return a pointer to it. Have `duplicate` return a null pointer if the memory allocation fails.

Section 17.3

3. Write the following function:

```
int *create_array(int n, int initial_value);
```

The function should return a pointer to a dynamically allocated `int` array with `n` members, each of which is initialized to `initial_value`. The return value should be `NULL` if the array can't be allocated.

Section 17.5

4. Suppose that the following declarations are in effect:

```
struct point { int x, y; };
struct rectangle { struct point upper_left, lower_right; };
struct rectangle *p;
```

Assume that we want `p` to point to a `rectangle` structure whose upper left corner is at (10, 25) and whose lower right corner is at (20, 15). Write a series of statements that allocate such a structure and initialize it as indicated.

W 5. Suppose that `f` and `p` are declared as follows:

```
struct {
  union {
    char a, b;
    int c;
  } d;
  int e[5];
} f, *p = &f;
```

Which of the following statements are legal?

(a) `p->b = ' ';`
(b) `p->e[3] = 10;`
(c) `(*p).d.a = '*';`
(d) `p->d->c = 20;`

6. Modify the `delete_from_list` function so that it uses only one pointer variable instead of two (`cur` and `prev`).

W 7. The following loop is supposed to delete all nodes from a linked list and release the memory that they occupy. Unfortunately, the loop is incorrect. Explain what's wrong with it and show how to fix the bug.

```
for (p = first; p != NULL; p = p->next)
  free(p);
```

W 8. Section 15.2 describes a file, `stack.c`, that provides functions for storing integers in a stack. In that section, the stack was implemented as an array. Modify `stack.c` so that a stack is now stored as a linked list. Replace the `contents` and `top` variables by a single variable that points to the first node in the list (the "top" of the stack). Write the functions in `stack.c` so that they use this pointer. Remove the `is_full` function, instead having `push` return either `true` (if memory was available to create a node) or `false` (if not).

9. True or false: If `x` is a structure and `a` is a member of that structure, then `(&x)->a` is the same as `x.a`. Justify your answer.

10. Modify the `print_part` function of Section 16.2 so that its parameter is a *pointer* to a `part` structure. Use the `->` operator in your answer.

11. Write the following function:

```
int count_occurrences(struct node *list, int n);
```

The `list` parameter points to a linked list; the function should return the number of times that `n` appears in this list. Assume that the `node` structure is the one defined in Section 17.5.

12. Write the following function:

```
struct node *find_last(struct node *list, int n);
```

The `list` parameter points to a linked list. The function should return a pointer to the *last* node that contains `n`; it should return `NULL` if `n` doesn't appear in the list. Assume that the `node` structure is the one defined in Section 17.5.

13. The following function is supposed to insert a new node into its proper place in an ordered list, returning a pointer to the first node in the modified list. Unfortunately, the function

doesn't work correctly in all cases. Explain what's wrong with it and show how to fix it. Assume that the node structure is the one defined in Section 17.5.

```
struct node *insert_into_ordered_list(struct node *list,
                                      struct node *new_node)
{
  struct node *cur = list, *prev = NULL;
  while (cur->value <= new_node->value) {
    prev = cur;
    cur = cur->next;
  }
  prev->next = new_node;
  new_node->next = cur;
  return list;
}
```

Section 17.6

14. Modify the `delete_from_list` function (Section 17.5) so that its first parameter has type `struct node **` (a pointer to a pointer to the first node in a list) and its return type is `void`. `delete_from_list` must modify its first argument to point to the list after the desired node has been deleted.

Section 17.7

W 15. Show the output of the following program and explain what it does.

```
#include <stdio.h>

int f1(int (*f)(int));
int f2(int i);

int main(void)
{
  printf("Answer: %d\n", f1(f2));
  return 0;
}

int f1(int (*f)(int))
{
  int n = 0;

  while ((*f)(n)) n++;
  return n;
}

int f2(int i)
{
  return i * i + i - 12;
}
```

16. Write the following function. The call `sum(g, i, j)` should return `g(i) + ... + g(j)`.

```
int sum(int (*f)(int), int start, int end);
```

W 17. Let `a` be an array of 100 integers. Write a call of `qsort` that sorts only the *last* 50 elements in `a`. (You don't need to write the comparison function).

18. Modify the `compare_parts` function so that parts are sorted with their numbers in *descending* order.

19. Write a function that, when given a string as its argument, searches the following array of structures for a matching command name, then calls the function associated with that name.

```
struct {
  char *cmd_name;
  void (*cmd_pointer)(void);
} file_cmd[] =
 {{"new",        new_cmd},
  {"open",       open_cmd},
  {"close",      close_cmd},
  {"close all", close_all_cmd},
  {"save",       save_cmd},
  {"save as",    save_as_cmd},
  {"save all",   save_all_cmd},
  {"print",      print_cmd},
  {"exit",       exit_cmd}
 };
```

Programming Projects

W 1. Modify the `inventory.c` program of Section 16.3 so that the `inventory` array is allocated dynamically and later reallocated when it fills up. Use `malloc` initially to allocate enough space for an array of 10 `part` structures. When the array has no more room for new parts, use `realloc` to double its size. Repeat the doubling step each time the array becomes full.

W 2. Modify the `inventory.c` program of Section 16.3 so that the `p` (print) command calls `qsort` to sort the `inventory` array before it prints the parts.

3. Modify the `inventory2.c` program of Section 17.5 by adding an `e` (erase) command that allows the user to remove a part from the database.

4. Modify the `justify` program of Section 15.3 by rewriting the `line.c` file so that it stores the current line in a linked list. Each node in the list will store a single word. The `line` array will be replaced by a variable that points to the node containing the first word. This variable will store a null pointer whenever the line is empty.

5. Write a program that sorts a series of words entered by the user:

```
Enter word: foo
Enter word: bar
Enter word: baz
Enter word: quux
Enter word:

In sorted order: bar baz foo quux
```

Assume that each word is no more than 20 characters long. Stop reading when the user enters an empty word (i.e., presses Enter without entering a word). Store each word in a dynamically allocated string, using an array of pointers to keep track of the strings, as in the `remind2.c` program (Section 17.2). After all words have been read, sort the array (using any sorting technique) and then use a loop to print the words in sorted order. *Hint:* Use the `read_line` function to read each word, as in `remind2.c`.

6. Modify Programming Project 5 so that it uses `qsort` to sort the array of pointers.

7. (C99) Modify the `remind2.c` program of Section 17.2 so that each element of the `reminders` array is a pointer to a `vstring` structure (see Section 17.9) rather than a pointer to an ordinary string.

18 Declarations

Making something variable is easy.
Controlling duration of constancy is the trick.

Declarations play a central role in C programming. By declaring variables and functions, we furnish vital information that the compiler will need in order to check a program for potential errors and translate it into object code.

Previous chapters have provided examples of declarations without going into full details; this chapter fills in the gaps. It explores the sophisticated options that can be used in declarations and reveals that variable declarations and function declarations have quite a bit in common. It also provides a firm grounding in the important concepts of storage duration, scope, and linkage.

Section 18.1 examines the syntax of declarations in their most general form, a topic that we've avoided up to this point. The next four sections focus on the items that appear in declarations: storage classes (Section 18.2), type qualifiers (Section 18.3), declarators (Section 18.4), and initializers (Section 18.5). Section 18.6 discusses the `inline` keyword, which can appear in C99 function declarations.

18.1 Declaration Syntax

Declarations furnish information to the compiler about the meaning of identifiers. When we write

```
int i;
```

we're informing the compiler that, in the current scope, the name `i` represents a variable of type `int`. The declaration

```
float f(float);
```

tells the compiler that `f` is a function that returns a `float` value and has one argument, also of type `float`.

In general, a declaration has the following appearance:

declaration *declaration-specifiers declarators* ;

Declaration specifiers describe the properties of the variables or functions being declared. ***Declarators*** give their names and may provide additional information about their properties.

Declaration specifiers fall into three categories:

- ***Storage classes.*** There are four storage classes: `auto`, `static`, `extern`, and `register`. At most one storage class may appear in a declaration; if present, it should come first.
- ***Type qualifiers.*** In C89, there are only two type qualifiers: `const` and `volatile`. C99 has a third type qualifier, `restrict`. A declaration may contain zero or more type qualifiers.

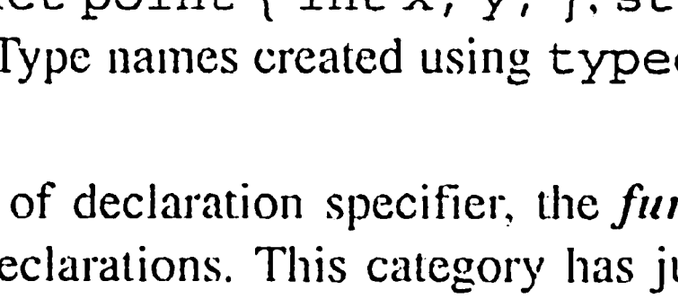
C99

- ***Type specifiers.*** The keywords `void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed`, and `unsigned` are all type specifiers. These words may be combined as described in Chapter 7; the order in which they appear doesn't matter (`int unsigned long` is the same as `long unsigned int`). Type specifiers also include specifications of structures, unions, and enumerations (for example, `struct point { int x, y; }`, `struct { int x, y; }`, or `struct point`). Type names created using `typedef` are type specifiers as well.

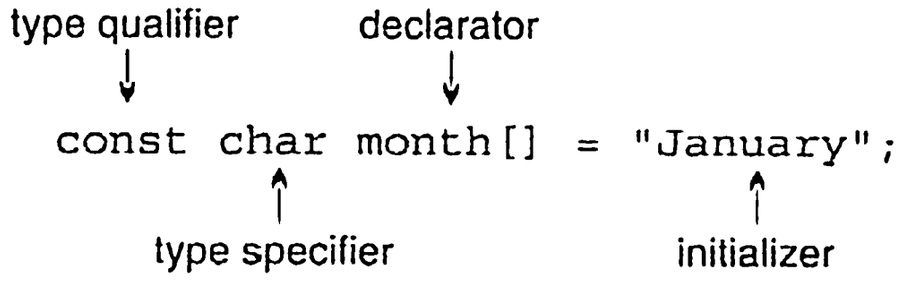
C99 (C99 has a fourth kind of declaration specifier, the ***function specifier***, which is used only in function declarations. This category has just one member, the keyword `inline`.) Type qualifiers and type specifiers should follow the storage class, but there are no other restrictions on their order. As a matter of style, I'll put type qualifiers before type specifiers.

Declarators include identifiers (names of simple variables), identifiers followed by `[]` (array names), identifiers preceded by `*` (pointer names), and identifiers followed by `()` (function names). Declarators are separated by commas. A declarator that represents a variable may be followed by an initializer.

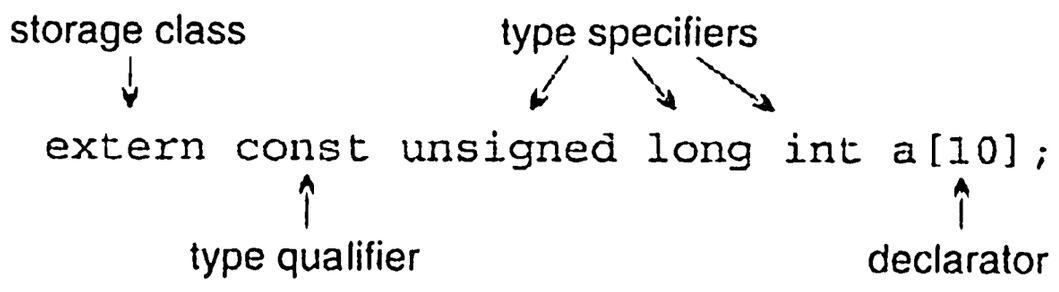
Let's look at a few examples that illustrate these rules. Here's a declaration with a storage class and three declarators:



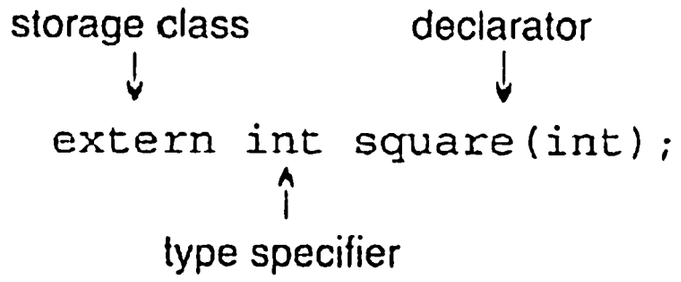
The following declaration has a type qualifier but no storage class. It also has an initializer:



The following declaration has both a storage class and a type qualifier. It also has three type specifiers; their order isn't important:



Function declarations, like variable declarations, may have a storage class, type qualifiers, and type specifiers. The following declaration has a storage class and a type specifier:



The next four sections cover storage classes, type qualifiers, declarators, and initializers in detail.

18.2 Storage Classes

Storage classes can be specified for variables and—to a lesser extent—functions and parameters. We'll concentrate on variables for now.

Recall from Section 10.3 that the term *block* refers to the body of a function (the part enclosed in braces) or a compound statement, possibly containing declarations. In C99, selection statements (`if` and `switch`) and iteration statements (`while`, `do`, and `for`)—along with the "inner" statements that they control—are considered to be blocks as well, although this is primarily a technicality.

C99

Q&A

Properties of Variables

Every variable in a C program has three properties:

- ***Storage duration.*** The storage duration of a variable determines when memory is set aside for the variable and when that memory is released. Storage for a variable with ***automatic storage duration*** is allocated when the surrounding

block is executed: storage is deallocated when the block terminates, causing the variable to lose its value. A variable with ***static storage duration*** stays at the same storage location as long as the program is running, allowing it to retain its value indefinitely.

- ***Scope.*** The scope of a variable is the portion of the program text in which the variable can be referenced. A variable can have either ***block scope*** (the variable is visible from its point of declaration to the end of the enclosing block) or ***file scope*** (the variable is visible from its point of declaration to the end of the enclosing file).
- ***Linkage.*** The linkage of a variable determines the extent to which it can be shared by different parts of a program. A variable with ***external linkage*** may be shared by several (perhaps all) files in a program. A variable with ***internal linkage*** is restricted to a single file, but may be shared by the functions in that file. (If a variable with the same name appears in another file, it's treated as a different variable.) A variable with ***no linkage*** belongs to a single function and can't be shared at all.

The default storage duration, scope, and linkage of a variable depend on where it's declared:

- Variables declared *inside* a block (including a function body) have *automatic* storage duration, *block* scope, and *no* linkage.
- Variables declared *outside* any block, at the outermost level of a program, have *static* storage duration, *file* scope, and *external* linkage.

The following example shows the default properties of the variables `i` and `j`:

```
int i;   /* static storage duration, file scope, external linkage */

void f(void)
{
  int j; /* automatic storage duration, block scope, no linkage */
}
```

For many variables, the default storage duration, scope, and linkage are satisfactory. When they aren't, we can alter these properties by specifying an explicit storage class: `auto`, `static`, `extern`, or `register`.

The `auto` Storage Class

The `auto` storage class is legal only for variables that belong to a block. An `auto` variable has automatic storage duration (not surprisingly), block scope, and no linkage. The `auto` storage class is almost never specified explicitly, since it's the default for variables declared inside a block.

The `static` Storage Class

The `static` storage class can be used with all variables, regardless of where they're declared, but it has a different effect on a variable declared outside a block than it does on a variable declared inside a block. When used *outside* a block, the word `static` specifies that a variable has internal linkage. When used *inside* a block, `static` changes the variable's storage duration from automatic to static. The following figure shows the effect of declaring `i` and `j` to be `static`:

```
static int i;      <- static storage duration, file scope, internal linkage

void f(void)
{
  static int j;    <- static storage duration, block scope, no linkage
}
```

When used in a declaration outside a block, `static` essentially hides a variable within the file in which it's declared: only functions that appear in the same file can see the variable. In the following example, the functions `f1` and `f2` both have access to `i`, but functions in other files don't:

```
static int i;

void f1(void)
{
  /* has access to i */
}

void f2(void)
{
  /* has access to i */
}
```

This use of `static` can help implement a technique known as information hiding.

information hiding ➤ 19.2

A `static` variable declared within a block resides at the same storage location throughout program execution. Unlike automatic variables, which lose their values each time the program leaves the enclosing block, a `static` variable will retain its value indefinitely. `static` variables have some interesting properties:

- A `static` variable in a block is initialized only once, prior to program execution. An `auto` variable is initialized every time it comes into existence (provided, of course, that it has an initializer).
- Each time a function is called recursively, it gets a new set of `auto` variables. If it has a `static` variable, on the other hand, that variable is shared by all calls of the function.

- Although a function shouldn't return a pointer to an `auto` variable, there's nothing wrong with it returning a pointer to a `static` variable.

Declaring one of its variables to be `static` allows a function to retain information between calls in a "hidden" area that the rest of the program can't access. More often, however, we'll use `static` to make programs more efficient. Consider the following function:

```
char digit_to_hex_char(int digit)
{
  const char hex_chars[16] = "0123456789ABCDEF";

  return hex_chars[digit];
}
```

Each time the `digit_to_hex_char` function is called, the characters `0123456789ABCDEF` will be copied into the `hex_chars` array to initialize it. Now, let's make the array `static`:

```
char digit_to_hex_char(int digit)
{
  static const char hex_chars[16] = "0123456789ABCDEF";

  return hex_chars[digit];
}
```

Since `static` variables are initialized only once, we've improved the speed of `digit_to_hex_char`.

The `extern` Storage Class

The `extern` storage class enables several source files to share the same variable. Section 15.2 covered the essentials of using `extern`, so I won't devote much space to it here. Recall that the declaration

```
extern int i;
```

informs the compiler that `i` is an `int` variable, but doesn't cause it to allocate memory for `i`. In C terminology, this declaration is not a *definition* of `i`; it merely informs the compiler that we need access to a variable that's defined elsewhere (perhaps later in the same file, or—more often—in another file). A variable can have many *declarations* in a program but should have only one *definition*.

There's one exception to the rule that an `extern` declaration of a variable isn't a definition. An `extern` declaration that initializes a variable serves as a definition of the variable. For example, the declaration

```
extern int i = 0;
```

is effectively the same as

```
int i = 0;
```

This rule prevents multiple `extern` declarations from initializing a variable in different ways.

A variable in an `extern` declaration always has static storage duration. The scope of the variable depends on the declaration's placement. If the declaration is inside a block, the variable has block scope; otherwise, it has file scope:

Q&A

```
                                  static storage duration
extern int i;                     file scope
                                  ? linkage

void f(void)
{
                                  static storage duration
  extern int j;                   block scope
                                  ? linkage
}
```

Determining the linkage of an `extern` variable is a bit harder. If the variable was declared `static` earlier in the file (outside of any function definition), then it has internal linkage. Otherwise (the normal case), the variable has external linkage.

The `register` Storage Class

Using the `register` storage class in the declaration of a variable asks the compiler to store the variable in a register instead of keeping it in main memory like other variables. (A *register* is a storage area located in a computer's CPU. Data stored in a register can be accessed and updated faster than data stored in ordinary memory.) Specifying the storage class of a variable to be `register` is a request, not a command. The compiler is free to store a `register` variable in memory if it chooses.

The `register` storage class is legal only for variables declared in a block. A `register` variable has the same storage duration, scope, and linkage as an `auto` variable. However, a `register` variable lacks one property that an `auto` variable has: since registers don't have addresses, it's illegal to use the `&` operator to take the address of a `register` variable. This restriction applies even if the compiler has elected to store the variable in memory.

`register` is best used for variables that are accessed and/or updated frequently. For example, the loop control variable in a `for` statement is a good candidate for `register` treatment:

```
int sum_array(int a[], int n)
{
  register int i;
  int sum = 0;

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

`register` isn't nearly as popular among C programmers as it once was. Today's compilers are much more sophisticated than early C compilers; many can determine automatically which variables would benefit the most from being kept in registers. Still, using `register` provides useful information that can help the compiler optimize the performance of a program. In particular, the compiler knows that a `register` variable can't have its address taken, and therefore can't be modified through a pointer. In this respect, the `register` keyword is related to C99's `restrict` keyword.

The Storage Class of a Function

Function declarations (and definitions), like variable declarations, may include a storage class, but the only options are `extern` and `static`. The word `extern` at the beginning of a function declaration specifies that the function has external linkage, allowing it to be called from other files. `static` indicates internal linkage, limiting use of the function's name to the file in which it's defined. If no storage class is specified, the function is assumed to have external linkage.

Consider the following function declarations:

```
extern int f(int i);
static int g(int i);
int h(int i);
```

`f` has external linkage, `g` has internal linkage, and `h` (by default) has external linkage. Because it has internal linkage, `g` can't be called directly from outside the file in which it's defined. (Declaring `g` to be `static` doesn't completely prevent it from being called in another file; an indirect call via a function pointer is still possible.)

Declaring functions to be `extern` is like declaring variables to be `auto`—it serves no purpose. For that reason, I don't use `extern` in function declarations. Be aware, however, that some programmers use `extern` extensively, which certainly does no harm.

Declaring functions to be `static`, on the other hand, is quite useful. In fact, I recommend using `static` when declaring any function that isn't intended to be called from other files. The benefits of doing so include:

- ***Easier maintenance.*** Declaring a function `f` to be `static` guarantees that `f` isn't visible outside the file in which its definition appears. As a result, someone modifying the program later knows that changes to `f` won't affect functions in other files. (One exception: a function in another file that's passed a pointer to `f` might be affected by changes to `f`. Fortunately, that situation is easy to spot by examining the file in which `f` is defined, since the function that passes `f` must also be defined there.)
- ***Reduced "name space pollution."*** Since functions declared `static` have internal linkage, their names can be reused in other files. Although we proba-

bly wouldn't deliberately reuse a function name for some other purpose, it can be hard to avoid in large programs. An excessive number of names with external linkage can result in what C programmers call "name space pollution": names in different files accidentally conflicting with each other. Using `static` helps prevent this problem.

Function parameters have the same properties as `auto` variables: automatic storage duration, block scope, and no linkage. The only storage class that can be specified for parameters is `register`.

Summary

Now that we've covered the various storage classes, let's summarize what we know. The following program fragment shows all possible ways to include—or omit—storage classes in declarations of variables and parameters.

```
int a;
extern int b;
static int c;

void f(int d, register int e)
{
  auto int g;
  int h;
  static int i;
  extern int j;
  register int k;
}
```

Table 18.1 shows the properties of each variable and parameter in this example.

Table 18.1
Properties of Variables and Parameters

Name	*Storage Duration*	*Scope*	*Linkage*
a	static	file	external
b	static	file	†
c	static	file	internal
d	automatic	block	none
e	automatic	block	none
g	automatic	block	none
h	automatic	block	none
i	static	block	none
j	static	block	†
k	automatic	block	none

†The definitions of b and j aren't shown, so it's not possible to determine the linkage of these variables. In most cases, the variables will be defined in another file and will have external linkage.

Of the four storage classes, the most important are `static` and `extern`. `auto` has no effect, and modern compilers have made `register` less important.

18.3 Type Qualifiers

C99

restricted pointers ➤ 17.8

There are two type qualifiers: `const` and `volatile`. (C99 has a third type qualifier, `restrict`, which is used only with pointers.) Since the use of `volatile` is limited to low-level programming, I'll postpone discussing it until Section 20.3.

`const` is used to declare objects that resemble variables but are "read-only": a program may access the value of a `const` object, but can't change it. For example, the declaration

```
const int n = 10;
```

creates a `const` object named `n` whose value is 10. The declaration

```
const int tax_brackets[] = {750, 2250, 3750, 5250, 7000};
```

creates a `const` array named `tax_brackets`.

Declaring an object to be `const` has several advantages:

- It's a form of documentation: it alerts anyone reading the program to the read-only nature of the object.
- The compiler can check that the program doesn't inadvertently attempt to change the value of the object.
- When programs are written for certain types of applications (embedded systems, in particular), the compiler can use the word `const` to identify data to be stored in ROM (read-only memory).

At first glance, it might appear that `const` serves the same role as the `#define` directive, which we've used in previous chapters to create names for constants. There are significant differences between `#define` and `const`, however:

- We can use `#define` to create a name for a numerical, character, or string constant. `const` can be used to create read-only objects of *any* type, including arrays, pointers, structures, and unions.
- `const` objects are subject to the same scope rules as variables; constants created using `#define` aren't. In particular, we can't use `#define` to create a constant with block scope.
- The value of a `const` object, unlike the value of a macro, can be viewed in a debugger.

Q&A

- Unlike macros, `const` objects can't be used in constant expressions. For example, we can't write

```
const int n = 10;
int a[n];               /*** WRONG ***/
```

C99

since array bounds must be constant expressions. (In C99, this example would

be legal if a has automatic storage duration—it would be treated as a variable-length array—but not if it has static storage duration.)

- It's legal to apply the address operator (&) to a `const` object, since it has an address. A macro doesn't have an address.

There are no absolute rules that dictate when to use `#define` and when to use `const`. I recommend using `#define` for constants that represent numbers or characters. That way, you'll be able to use the constants as array dimensions, in `switch` statements, and in other places where constant expressions are required.

18.4 Declarators

A declarator consists of an identifier (the name of the variable or function being declared), possibly preceded by the * symbol or followed by `[]` or `()`. By combining *, `[]`, and `()`, we can create declarators of mind-numbing complexity.

Before we look at the more complicated declarators, let's review the declarators that we've seen in previous chapters. In the simplest case, a declarator is just an identifier, like `i` in the following example:

```
int i;
```

Declarators may also contain the symbols *, `[]`, and `()`:

- A declarator that begins with * represents a pointer:

  ```
  int *p;
  ```

- A declarator that ends with `[]` represents an array:

  ```
  int a[10];
  ```

  The brackets may be left empty if the array is a parameter, if it has an initializer, or if its storage class is `extern`:

  ```
  extern int a[];
  ```

  Since `a` is defined elsewhere in the program, the compiler doesn't need to know its length here. (In the case of a multidimensional array, only the first set of brackets can be empty.) C99 provides two additional options for what goes between the brackets in the declaration of an array parameter. One option is the keyword `static`, followed by an expression that specifies the array's minimum length. The other is the * symbol, which can be used in a function prototype to indicate a variable-length array argument. Section 9.3 discusses both C99 features.

  C99

- A declarator that ends with `()` represents a function:

  ```
  int abs(int i);
  void swap(int *a, int *b);
  int find_largest(int a[], int n);
  ```

C allows parameter names to be omitted in a function declaration:

```
int abs(int);
void swap(int *, int *);
int find_largest(int [], int);
```

The parentheses can even be left empty:

```
int abs();
void swap();
int find_largest();
```

The declarations in the last group specify the return types of the `abs`, `swap`, and `find_largest` functions, but provide no information about their arguments. Leaving the parentheses empty isn't the same as putting the word `void` between them, which indicates that there are no arguments. The empty-parentheses style of function declaration has largely disappeared. It's inferior to the prototype style introduced in C89, since it doesn't allow the compiler to check whether function calls have the right arguments.

If all declarators were as simple as these, C programming would be a snap. Unfortunately, declarators in actual programs often combine the `*`, `[]`, and `()` notations. We've seen examples of such combinations already. We know that

```
int *ap[10];
```

declares an array of 10 pointers to integers. We know that

```
float *fp(float);
```

declares a function that has a `float` argument and returns a pointer to a `float`. And, in Section 17.7, we learned that

```
void (*pf)(int);
```

declares a pointer to a function with an `int` argument and a `void` return type.

Deciphering Complex Declarations

So far, we haven't had too much trouble understanding declarators. But what about declarators like the one in the following declaration?

```
int *(*x[10])(void);
```

This declarator combines `*`, `[]`, and `()`, so it's not obvious whether `x` is a pointer, an array, or a function.

Fortunately, there are two simple rules that will allow us to understand any declaration, no matter how convoluted:

- *Always read declarators from the inside out.* In other words, locate the identifier that's being declared, and start deciphering the declaration from there.

- ***When there's a choice, always favor*** `[]` ***and*** `()` ***over*** `*`. If `*` precedes the identifier and `[]` follows it, the identifier represents an array, not a pointer. Likewise, if `*` precedes the identifier and `()` follows it, the identifier represents a function, not a pointer. (Of course, we can always use parentheses to override the normal priority of `[]` and `()` over `*`.)

Let's apply these rules to our simple examples first. In the declaration

```
int *ap[10];
```

the identifier is `ap`. Since `*` precedes `ap` and `[]` follows it, we give preference to `[]`, so `ap` is an *array* of *pointers*. In the declaration

```
float *fp(float);
```

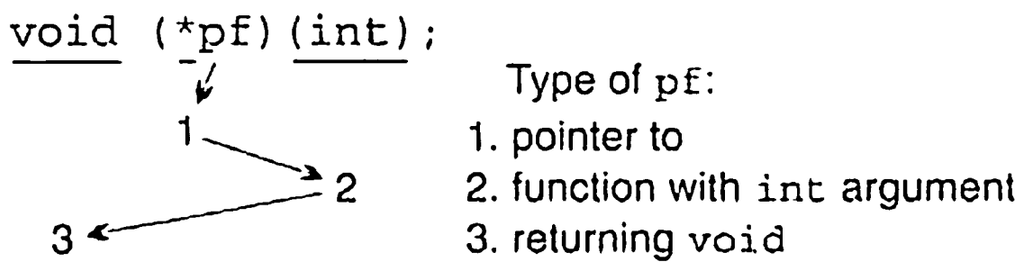
the identifier is `fp`. Since `*` precedes `fp` and `()` follows it, we give preference to `()`, so `fp` is a *function* that returns a *pointer*.

The declaration

```
void (*pf)(int);
```

is a little trickier. Since `*pf` is enclosed in parentheses, `pf` must be a pointer. But `(*pf)` is followed by `(int)`, so `pf` must point to a function with an `int` argument. The word `void` represents the return type of this function.

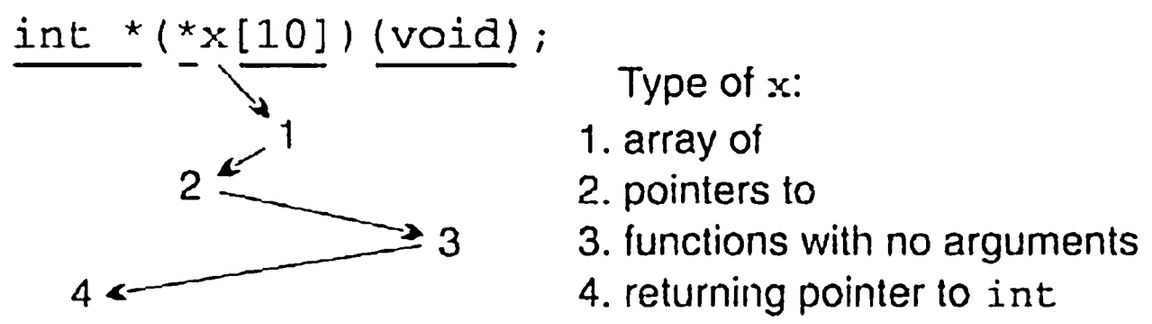
As the last example shows, understanding a complex declarator often involves zigzagging from one side of the identifier to the other:



Let's use this zigzagging technique to decipher the declaration given earlier:

```
int *(*x[10])(void);
```

First, we locate the identifier being declared (`x`). We see that `x` is preceded by `*` and followed by `[]`; since `[]` have priority over `*`, we go right (`x` is an array). Next, we go left to find out the type of the elements in the array (pointers). Next, we go right to find out what kind of data the pointers point to (functions with no arguments). Finally, we go left to see what each function returns (a pointer to an `int`). Graphically, here's what the process looks like:



Mastering C declarations takes time and practice. The only good news is that there are certain things that can't be declared in C. Functions can't return arrays:

```
int f(int)[];        /*** WRONG ***/
```

Functions can't return functions:

```
int g(int)(int);     /*** WRONG ***/
```

Arrays of functions aren't possible, either:

```
int a[10](int);      /*** WRONG ***/
```

In each case, we can use pointers to get the desired effect. A function can't return an array, but it can return a *pointer* to an array. A function can't return a function, but it can return a *pointer* to a function. Arrays of functions aren't allowed, but an array may contain *pointers* to functions. (Section 17.7 has an example of such an array.)

Using Type Definitions to Simplify Declarations

Some programmers use type definitions to help simplify complex declarations. Consider the declaration of x that we examined earlier in this section:

```
int *(*x[10])(void);
```

To make x's type easier to understand, we could use the following series of type definitions:

```
typedef int *Fcn(void);
typedef Fcn *Fcn_ptr;
typedef Fcn_ptr Fcn_ptr_array[10];
Fcn_ptr_array x;
```

If we read these lines in reverse order, we see that x has type `Fcn_ptr_array`, a `Fcn_ptr_array` is an array of `Fcn_ptr` values, a `Fcn_ptr` is a pointer to type `Fcn`, and a `Fcn` is a function that has no arguments and returns a pointer to an `int` value.

18.5 Initializers

For convenience, C allows us to specify initial values for variables as we're declaring them. To initialize a variable, we write the = symbol after its declarator, then follow that with an initializer. (Don't confuse the = symbol in a declaration with the assignment operator; initialization isn't the same as assignment.)

We've seen various kinds of initializers in previous chapters. The initializer for a simple variable is an expression of the same type as the variable:

```
int i = 5 / 2;    /* i is initially 2 */
```

If the types don't match, C converts the initializer using the same rules as for assignment:

conversion during assignment ➤7.4

```
int j = 5.5;      /* converted to 5 */
```

The initializer for a pointer variable must be a pointer expression of the same type as the variable or of type `void *`:

```
int *p = &i;
```

The initializer for an array, structure, or union is usually a series of values enclosed in braces:

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

C99

designated initializers ➤8.1, 16.1

In C99, brace-enclosed initializers can have other forms, thanks to designated initializers.

To complete our coverage of declarations, let's take a look at some additional rules that govern initializers:

- An initializer for a variable with static storage duration must be constant:

```
#define FIRST 1
#define LAST 100

static int i = LAST - FIRST + 1;
```

  Since `LAST` and `FIRST` are macros, the compiler can compute the initial value of `i` (100 – 1 + 1 = 100). If `LAST` and `FIRST` had been variables, the initializer would be illegal.

- If a variable has automatic storage duration, its initializer need not be constant:

```
int f(int n)
{
  int last = n - 1;
  ...
}
```

- A brace-enclosed initializer for an array, structure, or union must contain only constant expressions, never variables or function calls:

```
#define N 2

int powers[5] = {1, N, N * N, N * N * N, N * N * N * N};
```

  C99

  Since `N` is a constant, the initializer for `powers` is legal; if `N` were a variable, the program wouldn't compile. In C99, this restriction applies only if the variable has static storage duration.

- The initializer for an automatic structure or union can be another structure or union:

```
void g(struct part part1)
{
  struct part part2 = part1;
  ...
}
```

The initializer doesn't have to be a variable or parameter name, although it does need to be an expression of the proper type. For example, `part2`'s initializer could be `*p`, where `p` is of type `struct part *`, or `f(part1)`, where `f` is a function that returns a `part` structure.

Uninitialized Variables

In previous chapters, we've implied that uninitialized variables have undefined values. That's not always true; the initial value of a variable depends on its storage duration:

- Variables with *automatic* storage duration have no default initial value. The initial value of an automatic variable can't be predicted and may be different each time the variable comes into existence.
- Variables with *static* storage duration have the value zero by default. Unlike memory allocated by `calloc`, which is simply set to zero bits, a static variable is correctly initialized based on its type: integer variables are initialized to 0, floating variables are initialized to 0.0, and pointer variables contain a null pointer.

`calloc` function ➤ *17.3*

As a matter of style, it's better to provide initializers for static variables rather than rely on the fact that they're guaranteed to be zero. If a program accesses a variable that hasn't been initialized explicitly, someone reading the program later can't easily determine whether the variable is assumed to be zero or whether it's initialized by an assignment somewhere in the program.

18.6 Inline Functions (C99)

C99 function declarations have an additional option that doesn't exist in C89: they may contain the keyword `inline`. This keyword is a new breed of declaration specifier, distinct from storage classes, type qualifiers, and type specifiers. To understand the effect of `inline`, we'll need to visualize the machine instructions that are generated by a C compiler to handle the process of calling a function and returning from a function.

At the machine level, several instructions may need to be executed to prepare for the call, the call itself requires jumping to the first instruction in the function, and there may be additional instructions executed by the function itself as it begins to execute. If the function has arguments, they'll need to be copied (because C passes its arguments by value). Returning from a function requires a similar

amount of effort on both the part of the function that was called and the one that called it. The cumulative work required to call a function and later return from it is often referred to as "overhead," since it's extra work above and beyond what the function is really supposed to accomplish. Although the overhead of a function call slows the program by only a tiny amount, it may add up in certain situations, such as when a function is called millions or billions of times, when an older, slower processor is in use (as might be the case in an embedded system), or when a program has to meet very strict deadlines (as in a real-time system).

parameterized macros ➤ 14.3

In C89, the only way to avoid the overhead of a function call is to use a parameterized macro. Parameterized macros have certain drawbacks, though. C99 offers a better solution to this problem: create an *inline function*. The word "inline" suggests an implementation strategy in which the compiler replaces each call of the function by the machine instructions for the function. This technique avoids the usual overhead of a function call, although it may cause a minor increase in the size of the compiled program.

Declaring a function to be `inline` doesn't actually force the compiler to "inline" the function, however. It merely suggests that the compiler should try to make calls of the function as fast as possible, perhaps by performing an inline expansion when the function is called. The compiler is free to ignore this suggestion. In this respect, `inline` is similar to the `register` and `restrict` keywords, which the compiler may use to improve the performance of a program but may also choose to ignore.

Inline Definitions

An inline function has the keyword `inline` as one of its declaration specifiers:

```
inline double average(double a, double b)
{
  return (a + b) / 2;
}
```

Here's where things get a bit complicated. `average` has external linkage, so other source files may contain calls of `average`. However, the definition of `average` isn't considered to be an external definition by the compiler (it's an *inline definition* instead), so attempting to call `average` from another file will be considered an error.

There are two ways to avoid this error. One option is to add the word `static` to the function definition:

```
static inline double average(double a, double b)
{
  return (a + b) / 2;
}
```

`average` now has internal linkage, so it can't be called from other files. Other files may contain their own definitions of `average`, which might be the same as this definition or might be different.

The other option is to provide an external definition for `average` so that calls are permitted from other files. One way to do this is to write the `average` function a second time (without using `inline`) and put the second definition in a different source file. Doing so is legal, but it's not a good idea to have two versions of the same function, because we can't guarantee that they'll remain consistent when the program is modified.

Here's a better approach. First, we'll put the inline definition of `average` in a header file (let's name it `average.h`):

```
#ifndef AVERAGE_H
#define AVERAGE_H

inline double average(double a, double b)
{
  return (a + b) / 2;
}

#endif
```

Next, we'll create a matching source file, `average.c`:

```
#include "average.h"

extern double average(double a, double b);
```

Now, any file that needs to call the `average` function may simply include `average.h`, which contains the inline definition of `average`. The `average.c` file contains a prototype for `average` that uses the `extern` keyword, which causes the definition of `average` included from `average.h` to be treated as an external definition in `average.c`.

The general rule in C99 is that if all top-level declarations of a function in a particular file include `inline` but not `extern`, then the definition of the function in that file is inline. If the function is used anywhere in the program (including the file that contains its inline definition), then an external definition of the function will need to be provided by some other file. When the function is called, the compiler may choose to perform an ordinary call (using the function's external definition) or perform inline expansion (using the function's inline definition). There's no way to tell which choice the compiler will make, so it's crucial that the two definitions be consistent. The technique that we just discussed (using the `average.h` and `average.c` files) guarantees that the definitions are the same.

Restrictions on Inline Functions

Since inline functions are implemented in a way that's quite different from ordinary functions, they're subject to different rules and restrictions. Variables with static storage duration are a particular problem for inline functions with external linkage. Consequently, C99 imposes the following restrictions on an inline function with external linkage (but not on one with internal linkage):

- The function may not define a modifiable `static` variable.
- The function may not contain references to variables with internal linkage.

Such a function is allowed to define a variable that is both `static` and `const`, but each inline definition of the function may create its own copy of the variable.

Using Inline Functions with GCC

Some compilers, including GCC, supported inline functions prior to the C99 standard. As a result, their rules for using inline functions may vary from the standard. In particular, the scheme described earlier (using the `average.h` and `average.c` files) may not work with these compilers. Version 4.3 of GCC (not available at the time this book was written) is expected to support inline functions in the way described in the C99 standard.

Functions that are specified to be both `static` and `inline` should work fine, regardless of the version of GCC. This strategy is legal in C99 as well, so it's the safest bet. A `static inline` function can be used within a single file or placed in a header file and included into any source file that needs to call the function.

There's another way to share an inline function among multiple files that works with older versions of GCC but conflicts with C99. This technique involves putting a definition of the function in a header file, specifying that the function is both `extern` and `inline`, then including the header file into any source file that contains a call of the function. A second copy of the definition—without the words `extern` and `inline`—is placed in one of the source files. (That way, if the compiler is unable to "inline" the function for any reason, it will still have a definition.)

A final note about GCC: Functions are "inlined" only when optimization is requested via the `-O` command-line option.

Q & A

*Q: **Why are selection statements and iteration statements (and their "inner" statements) considered to be blocks in C99? [p. 459]**

C99

A: This rather surprising rule stems from a problem that can occur when compound literals are used in selection statements and iteration statements. The problem has to do with the storage duration of compound literals, so let's take a moment to discuss that issue first.

compound literals ➤9.3, 16.2

The C99 standard states that the object represented by a compound literal has static storage duration if the compound literal occurs outside the body of a function. Otherwise, it has automatic storage duration; as a result, the memory occupied by the object is deallocated at the end of the block in which the compound literal appears. Consider the following function, which returns a `point` structure created using a compound literal:

```
struct point create_point(int x, int y)
{
  return (struct point) {x, y};
}
```

This function works correctly, because the object created by the compound literal will be copied when the function returns. The original object will no longer exist, but the copy will remain. Now suppose that we change the function slightly:

```
struct point *create_point(int x, int y)
{
  return &(struct point) {x, y};
}
```

This version of `create_point` suffers from undefined behavior, because it returns a pointer to an object that has automatic storage duration and won't exist after the function returns.

Now let's return to the question we started with: Why are selection statements and iteration statements considered to be blocks? Consider the following example:

```
/* Example 1 - if statement without braces */

double *coefficients, value;

if (polynomial_selected == 1)
  coefficients = (double[3]) {1.5, -3.0, 6.0};
else
  coefficients = (double[3]) {4.5, 1.0, -3.5};
value = evaluate_polynomial(coefficients);
```

This program fragment apparently behaves in the desired fashion (but read on). `coefficients` will point to one of two objects created by compound literals. and this object will still exist at the time `evaluate_polynomial` is called. Now consider what happens if we put braces around the "inner" statements—the ones controlled by the `if` statement:

```
/* Example 2 - if statement with braces */

double *coefficients, value;

if (polynomial_selected == 1) {
  coefficients = (double[3]) {1.5, -3.0, 6.0};
} else {
  coefficients = (double[3]) {4.5, 1.0, -3.5};
}
value = evaluate_polynomial(coefficients);
```

Now we're in trouble. Each compound literal causes an object to be created, but that object exists only within the block formed by the braces that enclose the statement in which the literal appears. By the time `evaluate_polynomial` is called, `coefficients` points to an object that no longer exists. The result: undefined behavior.

The creators of C99 were unhappy with this state of affairs, because programmers were unlikely to expect that simply adding braces within an `if` statement would cause undefined behavior. To avoid the problem, they decided that the inner statements would always be considered blocks. As a result, Example 1 and Example 2 are equivalent, with both exhibiting undefined behavior.

A similar problem can arise when a compound literal is part of the controlling expression of a selection statement or iteration statement. For this reason, each entire selection statement and iteration statement is considered to be a block as well (as though an invisible set of braces surrounds the entire statement). So, for example, an `if` statement with an `else` clause consists of three blocks: each of the two inner statements is a block, as is the entire `if` statement.

Q: C99 You said that storage for a variable with automatic storage duration is allocated when the surrounding block is executed. Is this true for C99's variable-length arrays? [p. 460]

A: No. Storage for a variable-length array isn't allocated at the beginning of the surrounding block, because the length of the array isn't yet known. Instead, it's allocated when the declaration of the array is reached during the execution of the block. In this respect, variable-length arrays are different from all other automatic variables.

Q: What exactly is the difference between "scope" and "linkage"? [p. 460]

A: Scope is for the benefit of the compiler, while linkage is for the benefit of the linker. The compiler uses the scope of an identifier to determine whether or not it's legal to refer to the identifier at a given point in a file. When the compiler translates a source file into object code, it notes which names have external linkage, eventually storing these names in a table inside the object file. Thus, the linker has access to names with external linkage; names with internal linkage or no linkage are invisible to the linker.

Q: I don't understand how a name could have block scope but external linkage. Could you elaborate? [p. 463]

A: Certainly. Suppose that one source file defines a variable `i`:

```
int i;
```

Let's assume that the definition of `i` lies outside any function, so `i` has external linkage by default. In another file, there's a function `f` that needs to access `i`, so the body of `f` declares `i` as `extern`:

```
void f(void)
{
  extern int i;
  ...
}
```

In the first file, `i` has file scope. Within `f`, however, `i` has block scope. If other functions besides `f` need access to `i`, they'll need to declare it separately. (Or we

can simply move the declaration of `i` outside `f` so that `i` has file scope.) What's confusing about this entire business is that each declaration or definition of `i` establishes a different scope; sometimes it's file scope, and sometimes it's block scope.

***Q: Why can't `const` objects be used in constant expressions? `const` means "constant," right? [p. 466]**

A: In C, `const` means "read-only," not "constant." Let's look at a few examples that illustrate why `const` objects can't be used in constant expressions.

To start with, a `const` object might only be constant during its *lifetime*, not throughout the execution of the program. Suppose that a `const` object is declared inside a function:

```
void f(int n)
{
  const int m = n / 2;
  ...
}
```

When `f` is called, `m` will be initialized to the value of `n / 2`. The value of `m` will then remain constant until `f` returns. When `f` is called the next time, `m` will likely be given a different value. That's where the problem arises. Suppose that `m` appears in a `switch` statement:

```
void f(int n)
{
  const int m = n / 2;
  ...
  switch (...) {
    ...
    case m: ...   /*** WRONG ***/
    ...
  }
  ...
}
```

The value of `m` won't be known until `f` is called, which violates C's rule that the values of case labels must be constant expressions.

Next, let's look at `const` objects declared outside blocks. These objects have external linkage and can be shared among files. If C allowed the use of `const` objects in constant expressions, we could easily find ourselves in the following situation:

```
extern const int n;
int a[n];    /*** WRONG ***/
```

`n` is probably defined in another file, making it impossible for the compiler to determine `a`'s length. (I'm assuming that `a` is an external variable, so it can't be a variable-length array.)

volatile type qualifier ➤*20.3*

If that's not enough to convince you, consider this: If a `const` object is also declared to be `volatile`, its value may change at any time during execution. Here's an example from the C standard:

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

The `real_time_clock` variable may not be changed by the program (because it's declared `const`), yet its value may change via some other mechanism (because it's declared `volatile`).

Q: Why is the syntax of declarators so odd?

A: Declarations are intended to mimic use. A pointer declarator has the form `*p`, which matches the way the indirection operator will later be applied to `p`. An array declarator has the form `a[...]`, which matches the way the array will later be subscripted. A function declarator has the form `f(...)`, which matches the syntax of a function call. This reasoning extends to even the most complicated declarators. Consider the `file_cmd` array of Section 17.7, whose elements are pointers to functions. The declarator for `file_cmd` has the form

```
(*file_cmd[])(void)
```

and a call of one of the functions has the form

```
(*file_cmd[n])();
```

The parentheses, brackets, and * are in identical positions.

Exercises

Section 18.1

1. For each of the following declarations, identify the storage class, type qualifiers, type specifiers, declarators, and initializers.

   (a) `static char **lookup(int level);`
   (b) `volatile unsigned long io_flags;`
   (c) `extern char *file_name[MAX_FILES], path[];`
   (d) `static const char token_buf[] = "";`

Section 18.2

W 2. Answer each of the following questions with `auto`, `extern`, `register`, and/or `static`.

   (a) Which storage class is used primarily to indicate that a variable or function can be shared by several files?
   (b) Suppose that a variable `x` is to be shared by several functions in one file but hidden from functions in other files. Which storage class should `x` be declared to have?
   (c) Which storage classes can affect the storage duration of a variable?

3. List the storage duration (static or automatic), scope (block or file), and linkage (internal, external, or none) of each variable and parameter in the following file:

```
extern float a;

void f(register double b)
{
  static int c;
  auto char d;
}
```

W 4. Let `f` be the following function. What will be the value of `f(10)` if `f` has never been called before? What will be the value of `f(10)` if `f` has been called five times previously?

```
int f(int i)
{
  static int j = 0;
  return i * j++;
}
```

5. State whether each of the following statements is true or false. Justify each answer.
   (a) Every variable with static storage duration has file scope.
   (b) Every variable declared inside a function has no linkage.
   (c) Every variable with internal linkage has static storage duration.
   (d) Every parameter has block scope.

6. The following function is supposed to print an error message. Each message is preceded by an integer, indicating the number of times the function has been called. Unfortunately, the function always displays 1 as the number of the error message. Locate the error and show how to fix it without making any changes outside the function.

```
void print_error(const char *message)
{
  int n = 1;
  printf("Error %d: %s\n", n++, message);
}
```

Section 18.3

7. Suppose that we declare `x` to be a `const` object. Which one of the following statements about `x` is *false*?
   (a) If `x` is of type `int`, it can be used as the value of a case label in a `switch` statement.
   (b) The compiler will check that no assignment is made to `x`.
   (c) `x` is subject to the same scope rules as variables.
   (d) `x` can be of any type.

Section 18.4

W 8. Write a complete description of the type of `x` as specified by each of the following declarations.
   (a) `char (*x[10])(int);`
   (b) `int (*x(int))[5];`
   (c) `float *(*x(void))(int);`
   (d) `void (*x(int, void (*y)(int)))(int);`

9. Use a series of type definitions to simplify each of the declarations in Exercise 8.

W 10. Write declarations for the following variables and functions:
   (a) `p` is a pointer to a function with a character pointer argument that returns a character pointer.

(b) `f` is a function with two arguments: `p`, a pointer to a structure with tag `t`, and `n`, a long integer. `f` returns a pointer to a function that has no arguments and returns nothing.
(c) `a` is an array of four pointers to functions that have no arguments and return nothing. The elements of `a` initially point to functions named `insert`, `search`, `update`, and `print`.
(d) `b` is an array of 10 pointers to functions with two `int` arguments that return structures with tag `t`.

11. In Section 18.4, we saw that the following declarations are illegal:

```
int f(int)[];       /* functions can't return arrays     */
int g(int)(int);    /* functions can't return functions  */
int a[10](int);     /* array elements can't be functions */
```

We can, however, achieve similar effects by using pointers: a function can return a *pointer* to the first element in an array, a function can return a *pointer* to a function, and the elements of an array can be *pointers* to functions. Revise each of these declarations accordingly.

*12. (a) Write a complete description of the type of the function `f`, assuming that it's declared as follows:

```
int (*f(float (*)(long), char *))(double);
```

(b) Give an example showing how `f` would be called.

Section 18.5

W 13. Which of the following declarations are legal? (Assume that `PI` is a macro that represents 3.14159.)

```
(a) char c = 65;
(b) static int i = 5, j = i * i;
(c) double d = 2 * PI;
(d) double angles[] = {0, PI / 2, PI, 3 * PI / 2};
```

14. Which kind of variables cannot be initialized?

(a) Array variables
(b) Enumeration variables
(c) Structure variables
(d) Union variables
(e) None of the above

W 15. Which property of a variable determines whether or not it has a default initial value?

(a) Storage duration
(b) Scope
(c) Linkage
(d) Type

19 Program Design

Wherever there is modularity there is the potential for misunderstanding: Hiding information implies a need to check communication.

It's obvious that real-world programs are larger than the examples in this book, but you may not realize just how much larger. Faster CPUs and larger main memories have made it possible to write programs that would have been impractical just a few years ago. The popularity of graphical user interfaces has added greatly to the average length of a program. Most full-featured programs today are at least 100,000 lines long. Million-line programs are commonplace, and it's not unheard-of for a program to have 10 million lines or more.

Q&A

Although C wasn't designed for writing large programs, many large programs have in fact been written in C. It's tricky, and it requires a great deal of care, but it can be done. In this chapter, I'll discuss techniques that have proved to be helpful for writing large programs and show which C features (the `static` storage class, for example) are especially useful.

Writing large programs (often called "programming-in-the-large") is quite different from writing small ones—it's like the difference between writing a term paper (10 pages double-spaced, of course) and a 1000-page book. A large program requires more attention to style, since many people will be working on it. It requires careful documentation. It requires planning for maintenance, since it will likely be modified many times.

Above all, a large program requires careful design and much more planning than a small program. As Alan Kay, the designer of the Smalltalk programming language, puts it, "You can build a doghouse out of anything." A doghouse can be built without any particular design, using whatever materials are at hand. A house for humans, on the other hand, is too complex to just throw together.

Chapter 15 discussed writing large programs in C, but it concentrated on language details. In this chapter, we'll revisit the topic, this time focusing on techniques for good program design. A complete discussion of program design issues is obviously beyond the scope of this book. However, I'll try to cover—briefly—

some important concepts in program design and show how to use them to create C programs that are readable and maintainable.

Section 19.1 discusses how to view a C program as a collection of modules that provide services to each other. We'll then see how the concepts of information hiding (Section 19.2) and abstract data types (Section 19.3) can improve modules. By focusing on a single example (a stack data type), Section 19.4 illustrates how an abstract data type can be defined and implemented in C. Section 19.5 describes some limitations of C for defining abstract data types and shows how to work around them.

19.1 Modules

When designing a C program (or a program in any other language, for that matter), it's often useful to view it as a number of independent ***modules.*** A module is a collection of services, some of which are made available to other parts of the program (the ***clients***). Each module has an ***interface*** that describes the available services. The details of the module—including the source code for the services themselves—are stored in the module's ***implementation***.

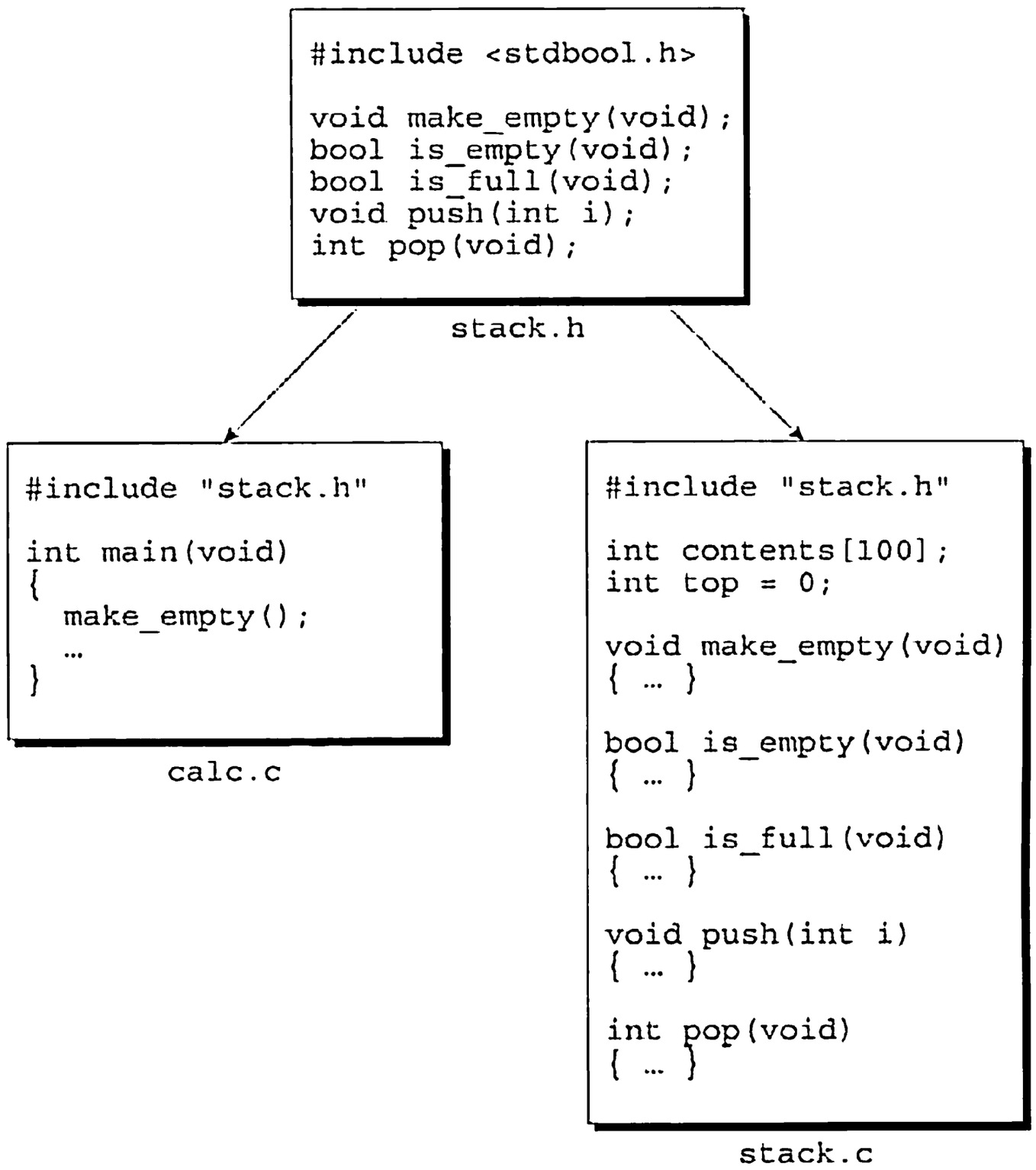
In the context of C, "services" are functions. The interface of a module is a header file containing prototypes for the functions that will be made available to clients (source files). The implementation of a module is a source file that contains definitions of the module's functions.

To illustrate this terminology, let's look at the calculator program that was sketched in Sections 15.1 and 15.2. This program consists of the file `calc.c`, which contains the `main` function, and a stack module, which is stored in the files `stack.h` and `stack.c` (see the figure at the top of the next page). `calc.c` is a *client* of the stack module. `stack.h` is the *interface* of the stack module; it supplies everything the client needs to know about the module. `stack.c` is the *implementation* of the module; it contains definitions of the stack functions as well as declarations of the variables that make up the stack.

The C library is itself a collection of modules. Each header in the library serves as the interface to a module. `<stdio.h>`, for example, is the interface to a module containing I/O functions, while `<string.h>` is the interface to a module containing string-handling functions.

Dividing a program into modules has several advantages:

- ***Abstraction.*** If modules are properly designed, we can treat them as ***abstractions***; we know what they do, but we don't worry about the details of how they do it. Thanks to abstraction, it's not necessary to understand how the entire program works in order to make changes to one part of it. What's more, abstraction makes it easier for several members of a team to work on the same program. Once the interfaces for the modules have been agreed upon, the responsibility for implementing each module can be delegated to a partic-



ular person. Team members can then work largely independently of one another.

- ***Reusability.*** Any module that provides services is potentially reusable in other programs. Our stack module, for example, is reusable. Since it's often hard to anticipate the future uses of a module, it's a good idea to design modules for reusability.
- ***Maintainability.*** A small bug will usually affect only a single module implementation, making the bug easier to locate and fix. Once the bug has been fixed, rebuilding the program requires only a recompilation of the module implementation (followed by linking the entire program). On a larger scale, we could replace an entire module implementation, perhaps to improve performance or when transporting the program to a different platform.

Although all these advantages are important, maintainability is the most critical. Most real-world programs are in service over a period of years, during which bugs are discovered, enhancements are made, and modifications are made to meet changing requirements. Designing a program in a modular fashion makes maintenance much easier. Maintaining a program should be like maintaining a car—fixing a flat tire shouldn't require overhauling the engine.

For an example, we need look no further than the `inventory` program of Chapters 16 and 17. The original program (Section 16.3) stored part records in an array. Suppose that, after using this program for a while, the customer objects to having a fixed limit on the number of parts that can be stored. To satisfy the customer, we might switch to a linked list (as we did in Section 17.5). Making this change required going through the entire program, looking for all places that depend on the way parts are stored. If we'd designed the program differently in the first place—with a separate module dealing with part storage—we would have only needed to rewrite the implementation of that module, not the entire program.

Once we're convinced that modular design is the way to go, the process of designing a program boils down to deciding what modules it should have, what services each module should provide, and how the modules should be interrelated. We'll now look at these issues briefly. For more information about design, consult a software engineering text, such as *Fundamentals of Software Engineering*, Second Edition, by Ghezzi, Jazayeri, and Mandrioli (Upper Saddle River, N.J.: Prentice-Hall, 2003).

Cohesion and Coupling

Good module interfaces aren't random collections of declarations. In a well-designed program, modules should have two properties:

- ***High cohesion.*** The elements of each module should be closely related to one another; we might think of them as cooperating toward a common goal. High cohesion makes modules easier to use and makes the entire program easier to understand.
- ***Low coupling.*** Modules should be as independent of each other as possible. Low coupling makes it easier to modify the program and reuse modules.

Does the calculator program have these properties? The stack module is clearly cohesive: its functions represent operations on a stack. There's little coupling in the program. The `calc.c` file depends on `stack.h` (and `stack.c` depends on `stack.h`, of course), but there are no other apparent dependencies.

Types of Modules

Because of the need for high cohesion and low coupling, modules tend to fall into certain typical categories:

- A ***data pool*** is a collection of related variables and/or constants. In C, a module of this type is often just a header file. From a design standpoint, putting variables in header files isn't usually a good idea, but collecting related constants in a header file can often be useful. In the C library, `<float.h>` and `<limits.h>` are both data pools.

`<float.h>` header ➤23.1
`<limits.h>` header ➤23.2

- A ***library*** is a collection of related functions. The `<string.h>` header, for example, is the interface to a library of string-handling functions.

- An *abstract object* is a collection of functions that operate on a hidden data structure. (In this chapter, the term "object" has a different meaning than in the rest of the book. In C terminology, an object is simply a block of memory that can store a value. In this chapter, however, an object is a collection of data bundled with operations on the data. If the data is hidden, the object is "abstract.") The stack module we've been discussing belongs to this category.
- An *abstract data type (ADT)* is a type whose representation is hidden. Client modules can use the type to declare variables, but have no knowledge of the structure of those variables. For a client module to perform an operation on such a variable, it must call a function provided by the abstract data type module. Abstract data types play a significant role in modern programming; we'll return to them in Sections 19.3–19.5.

19.2 Information Hiding

A well-designed module often keeps some information secret from its clients. Clients of our stack module, for example, have no need to know whether the stack is stored in an array, in a linked list, or in some other form. Deliberately concealing information from the clients of a module is known as *information hiding*. Information hiding has two primary advantages:

- *Security.* If clients don't know how the stack is stored, they won't be able to corrupt it by tampering with its internal workings. To perform operations on the stack, they'll have to call functions that are provided by the module itself—functions that we've written and tested.
- *Flexibility.* Making changes—no matter how large—to a module's internal workings won't be difficult. For example, we could implement the stack as an array at first, then later switch to a linked list or other representation. We'll have to rewrite the implementation of the module, of course, but—if the module was designed properly—we won't have to alter the module's interface.

`static` storage class ➤ *18.2*

In C, the major tool for enforcing information hiding is the `static` storage class. Declaring a variable with file scope to be `static` gives it internal linkage, thus preventing it from being accessed from other files, including clients of the module. (Declaring a function to be `static` is also useful—the function can be directly called only by other functions in the same file.)

A Stack Module

To see the benefits of information hiding, let's look at two implementations of a stack module, one using an array and the other a linked list. The module's header file will have the following appearance:

stack.h

```
#ifndef STACK_H
#define STACK_H

#include <stdbool.h>   /* C99 only */

void make_empty(void);
bool is_empty(void);
bool is_full(void);
void push(int i);
int pop(void);

#endif
```

I've included C99's `<stdbool.h>` header so that the `is_empty` and `is_full` functions can return a `bool` result rather than an `int` value.

Let's first use an array to implement the stack:

stack1.c

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

#define STACK_SIZE 100

static int contents[STACK_SIZE];
static int top = 0;

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

void make_empty(void)
{
  top = 0;
}

bool is_empty(void)
{
  return top == 0;
}

bool is_full(void)
{
  return top == STACK_SIZE;
}

void push(int i)
{
  if (is_full())
    terminate("Error in push: stack is full.");
  contents[top++] = i;
}
```

```
int pop(void)
{
  if (is_empty())
    terminate("Error in pop: stack is empty.");
  return contents[--top];
}
```

The variables that make up the stack (`contents` and `top`) are both declared `static`, since there's no reason for the rest of the program to access them directly. The `terminate` function is also declared `static`. This function isn't part of the module's interface; instead, it's designed for use solely within the implementation of the module.

As a matter of style, some programmers use macros to indicate which functions and variables are "public" (accessible elsewhere in the program) and which are "private" (limited to a single file):

```
#define PUBLIC  /* empty */
#define PRIVATE static
```

The reason for writing `PRIVATE` instead of `static` is that the latter has more than one use in C; `PRIVATE` makes it clear that we're using it to enforce information hiding. Here's what the stack implementation would look like if we were to use `PUBLIC` and `PRIVATE`:

```
PRIVATE int contents[STACK_SIZE];
PRIVATE int top = 0;

PRIVATE void terminate(const char *message) { ... }

PUBLIC void make_empty(void) { ... }

PUBLIC bool is_empty(void) { ... }

PUBLIC bool is_full(void) { ... }

PUBLIC void push(int i) { ... }

PUBLIC int pop(void) { ... }
```

Now we'll switch to a linked-list implementation of the stack module:

stack2.c

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

struct node {
  int data;
  struct node *next;
};

static struct node *top = NULL;
```

```
static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

void make_empty(void)
{
  while (!is_empty())
    pop();
}

bool is_empty(void)
{
  return top == NULL;
}

bool is_full(void)
{
  return false;
}

void push(int i)
{
  struct node *new_node = malloc(sizeof(struct node));
  if (new_node == NULL)
    terminate("Error in push: stack is full.");

  new_node->data = i;
  new_node->next = top;
  top = new_node;
}

int pop(void)
{
  struct node *old_top;
  int i;

  if (is_empty())
    terminate("Error in pop: stack is empty.");

  old_top = top;
  i = top->data;
  top = top->next;
  free(old_top);
  return i;
}
```

Note that the `is_full` function returns `false` every time it's called. A linked list has no limit on its size, so the stack will never be full. It's possible (but not likely) that the program might run out of memory, which will cause the `push` function to fail, but there's no easy way to test for that condition in advance.

Our stack example shows clearly the advantage of information hiding: it

doesn't matter whether we use `stack1.c` or `stack2.c` to implement the stack module. Both versions match the module's interface, so we can switch from one to the other without having to make changes elsewhere in the program.

19.3 Abstract Data Types

A module that serves as an abstract object, like the stack module in the previous section, has a serious disadvantage: there's no way to have multiple instances of the object (more than one stack, in this case). To accomplish this, we'll need to go a step further and create a new *type*.

Once we've defined a `Stack` type, we'll be able to have as many stacks as we want. The following fragment illustrates how we could have two stacks in the same program:

```
Stack s1, s2;

make_empty(&s1);
make_empty(&s2);
push(&s1, 1);
push(&s2, 2);
if (!is_empty(&s1))
  printf("%d\n", pop(&s1));    /* prints "1" */
```

We're not really sure what `s1` and `s2` are (structures? pointers?), but it doesn't matter. To clients, `s1` and `s2` are *abstractions* that respond to certain operations (`make_empty`, `is_empty`, `is_full`, `push`, and `pop`).

Let's convert our `stack.h` header so that it provides a `Stack` type, where `Stack` is a structure. Doing so will require adding a `Stack` (or `Stack *`) parameter to each function. The header will now look like this (changes to `stack.h` are in **bold**; unchanged portions of the header aren't shown):

```
#define STACK_SIZE 100

typedef struct {
  int contents[STACK_SIZE];
  int top;
} Stack;

void make_empty(Stack *s);
bool is_empty(const Stack *s);
bool is_full(const Stack *s);
void push(Stack *s, int i);
int pop(Stack *s);
```

The stack parameters to `make_empty`, `push`, and `pop` need to be pointers, since these functions modify the stack. The parameter to `is_empty` and `is_full` doesn't need to be a pointer, but I've made it one anyway. Passing these functions a `Stack` *pointer* instead of a `Stack` *value* is more efficient, since the latter would result in a structure being copied.

Encapsulation

Unfortunately, `Stack` isn't an *abstract* data type, since `stack.h` reveals what the `Stack` type really is. Nothing prevents clients from using a `Stack` variable as a structure:

```
Stack s1;

s1.top = 0;
s1.contents[top++] = 1;
```

Providing access to the `top` and `contents` members allows clients to corrupt the stack. Worse still, we won't be able to change the way stacks are stored without having to assess the effect of the change on clients.

What we need is a way to prevent clients from knowing how the `Stack` type is represented. C has only limited support for ***encapsulating*** types in this way. Newer C-based languages, including C++, Java, and C#, are better equipped for this purpose.

Incomplete Types

The only tool that C gives us for encapsulation is the ***incomplete type***. (Incomplete types were mentioned briefly in Section 17.9 and in the Q&A section at the end of Chapter 17.) The C standard describes incomplete types as "types that describe objects but lack information needed to determine their sizes." For example, the declaration

Q&A

```
struct t;   /* incomplete declaration of t */
```

tells the compiler that `t` is a structure tag but doesn't describe the members of the structure. As a result, the compiler doesn't have enough information to determine the size of such a structure. The intent is that an incomplete type will be completed elsewhere in the program.

As long as a type remains incomplete, its uses are limited. Since the compiler doesn't know the size of an incomplete type, it can't be used to declare a variable:

Q&A

```
struct t s;   /*** WRONG ***/
```

However, it's perfectly legal to define a pointer type that references an incomplete type:

```
typedef struct t *T;
```

This type definition states that a variable of type `T` is a pointer to a structure with tag `t`. We can now declare variables of type `T`, pass them as arguments to functions, and perform other operations that are legal for pointers. (The size of a pointer doesn't depend on what it points to, which explains why C allows this behavior.) What we can't do, though, is apply the `->` operator to one of these variables, since the compiler knows nothing about the members of a `t` structure.

19.4 A Stack Abstract Data Type

To illustrate how abstract data types can be encapsulated using incomplete types, we'll develop a stack ADT based on the stack module described in Section 19.2. In the process, we'll explore three different ways to implement the stack.

Defining the Interface for the Stack ADT

First, we'll need a header file that defines our stack ADT type and gives prototypes for the functions that represent stack operations. Let's name this file `stackADT.h`. The `Stack` type will be a pointer to a `stack_type` structure that stores the actual contents of the stack. This structure is an incomplete type that will be completed in the file that implements the stack. The members of this structure will depend on how the stack is implemented. Here's what the `stackADT.h` file will look like:

stackADT.h (version 1)

```
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>   /* C99 only */

typedef struct stack_type *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);
void push(Stack s, int i);
int pop(Stack s);

#endif
```

Clients that include `stackADT.h` will be able to declare variables of type `Stack`, each of which is capable of pointing to a `stack_type` structure. Clients can then call the functions declared in `stackADT.h` to perform operations on stack variables. However, clients can't access the members of the `stack_type` structure, since that structure will be defined in a separate file.

Note that each function has a `Stack` parameter or returns a `Stack` value. The stack functions in Section 19.3 had parameters of type `Stack *`. The reason for the difference is that a `Stack` variable is now a pointer; it points to a `stack_type` structure that stores the contents of the stack. If a function needs to modify the stack, it changes the structure itself, not the pointer to the structure.

Also note the presence of the `create` and `destroy` functions. A module

generally doesn't need these functions, but an ADT does. `create` will dynamically allocate memory for a stack (including the memory required for a `stack_type` structure), as well as initializing the stack to its "empty" state. `destroy` will release the stack's dynamically allocated memory.

The following client file can be used to test the stack ADT. It creates two stacks and performs a variety of operations on them.

stackclient.c

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

int main(void)
{
  Stack s1, s2;
  int n;

  s1 = create();
  s2 = create();

  push(s1, 1);
  push(s1, 2);

  n = pop(s1);
  printf("Popped %d from s1\n", n);
  push(s2, n);
  n = pop(s1);
  printf("Popped %d from s1\n", n);
  push(s2, n);

  destroy(s1);

  while (!is_empty(s2))
    printf("Popped %d from s2\n", pop(s2));

  push(s2, 3);
  make_empty(s2);
  if (is_empty(s2))
    printf("s2 is empty\n");
  else
    printf("s2 is not empty\n");

  destroy(s2);

  return 0;
}
```

If the stack ADT is implemented correctly, the program should produce the following output:

```
Popped 2 from s1
Popped 1 from s1
Popped 1 from s2
Popped 2 from s2
s2 is empty
```

Implementing the Stack ADT Using a Fixed-Length Array

There are several ways to implement the stack ADT. Our first approach is the simplest. We'll have the `stackADT.c` file define the `stack_type` structure so that it contains a fixed-length array (to hold the contents of the stack) along with an integer that keeps track of the top of the stack:

```
struct stack_type {
  int contents[STACK_SIZE];
  int top;
};
```

Here's what `stackADT.c` will look like:

stackADT.c

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

#define STACK_SIZE 100

struct stack_type {
  int contents[STACK_SIZE];
  int top;
};

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

Stack create(void)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL)
    terminate("Error in create: stack could not be created.");
  s->top = 0;
  return s;
}

void destroy(Stack s)
{
  free(s);
}

void make_empty(Stack s)
{
  s->top = 0;
}

bool is_empty(Stack s)
{
  return s->top == 0;
}
```

```
bool is_full(Stack s)
{
  return s->top == STACK_SIZE;
}

void push(Stack s, int i)
{
  if (is_full(s))
    terminate("Error in push: stack is full.");
  s->contents[s->top++] = i;
}

int pop(Stack s)
{
  if (is_empty(s))
    terminate("Error in pop: stack is empty.");
  return s->contents[--s->top];
}
```

The most striking thing about the functions in this file is that they use the `->` operator, not the `.` operator, to access the `contents` and `top` members of the `stack_type` structure. The `s` parameter is a pointer to a `stack_type` structure, not a structure itself, so using the `.` operator would be illegal.

Changing the Item Type in the Stack ADT

Now that we have a working version of the stack ADT, let's try to improve it. First, note that items in the stack must be integers. That's too restrictive; in fact, the item type doesn't really matter. The stack items could just as easily be other basic types (`float`, `double`, `long`, etc.) or even structures, unions, or pointers, for that matter.

To make the stack ADT easier to modify for different item types, let's add a type definition to the `stackADT.h` header. It will define a type named `Item`, representing the type of data to be stored on the stack.

stackADT.h (version 2)

```
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>   /* C99 only */

typedef int Item;

typedef struct stack_type *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);
```

```
void push(Stack s, Item i);
Item pop(Stack s);

#endif
```

The changes to the file are shown in **bold.** Besides the addition of the `Item` type, the `push` and `pop` functions have been modified. `push` now has a parameter of type `Item`, and `pop` returns a value of type `Item`. We'll use this version of `stackADT.h` from now on; it replaces the earlier version.

The `stackADT.c` file will need to be modified to match the new `stackADT.h`. The changes are minimal, however. The `stack_type` structure will now contain an array whose elements have type `Item` instead of `int`:

```
struct stack_type {
  Item contents[STACK_SIZE];
  int top;
};
```

The only other changes are to `push` (the second parameter now has type `Item`) and `pop` (which returns a value of type `Item`). The bodies of `push` and `pop` are unchanged.

The `stackclient.c` file can be used to test the new `stackADT.h` and `stackADT.c` to verify that the `Stack` type still works (it does!). Now we can change the item type any time we want by simply modifying the definition of the `Item` type in `stackADT.h`. (Although we won't have to change the `stackADT.c` file, we'll still need to recompile it.)

Implementing the Stack ADT Using a Dynamic Array

Another problem with the stack ADT as it currently stands is that each stack has a fixed maximum size, which is currently set at 100 items. This limit can be increased to any number we wish, of course, but all stacks created using the `Stack` type will have the same limit. There's no way to have stacks with different capacities or to set the stack size as the program is running.

There are two solutions to this problem. One is to implement the stack as a linked list, in which case there's no fixed limit on its size. We'll investigate this solution in a moment. First, though, let's try the other approach, which involves storing stack items in a dynamically allocated array.

dynamically allocated arrays ➤ 17.3

The crux of the latter approach is to modify the `stack_type` structure so that the `contents` member is a *pointer* to the array in which the items are stored, not the array itself:

```
struct stack_type {
  Item *contents;
  int top;
  int size;
};
```

I've also added a new member, `size`, that stores the stack's maximum size (the length of the array that `contents` points to). We'll use this member to check for the "stack full" condition.

The `create` function will now have a parameter that specifies the desired maximum stack size:

```
Stack create(int size);
```

When `create` is called, it will create a `stack_type` structure plus an array of length `size`. The `contents` member of the structure will point to this array.

The `stackADT.h` file will be the same as before, except that we'll need to add a `size` parameter to the `create` function. (Let's name the new version `stackADT2.h`.) The `stackADT.c` file will need more extensive modification, however. The new version appears below, with changes shown in **bold.**

stackADT2.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stackADT2.h"

struct stack_type {
  Item *contents;
  int top;
  int size;
};

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

Stack create(int size)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL)
    terminate("Error in create: stack could not be created.");
  s->contents = malloc(size * sizeof(Item));
  if (s->contents == NULL) {
    free(s);
    terminate("Error in create: stack could not be created.");
  }
  s->top = 0;
  s->size = size;
  return s;
}

void destroy(Stack s)
{
  free(s->contents);
  free(s);
}
```

```
void make_empty(Stack s)
{
  s->top = 0;
}

bool is_empty(Stack s)
{
  return s->top == 0;
}

bool is_full(Stack s)
{
  return s->top == s->size;
}

void push(Stack s, Item i)
{
  if (is_full(s))
    terminate("Error in push: stack is full.");
  s->contents[s->top++] = i;
}

Item pop(Stack s)
{
  if (is_empty(s))
    terminate("Error in pop: stack is empty.");
  return s->contents[--s->top];
}
```

The `create` function now calls `malloc` twice: once to allocate a `stack_type` structure and once to allocate the array that will contain the stack items. Either call of `malloc` could fail, causing `terminate` to be called. The `destroy` function must call `free` twice to release all the memory allocated by `create`.

The `stackclient.c` file can again be used to test the stack ADT. The calls of `create` will need to be changed, however, since `create` now requires an argument. For example, we could replace the statements

```
s1 = create();
s2 = create();
```

with the following statements:

```
s1 = create(100);
s2 = create(200);
```

Implementing the Stack ADT Using a Linked List

Implementing the stack ADT using a dynamically allocated array gives us more flexibility than using a fixed-size array. However, the client is still required to specify a maximum size for a stack at the time it's created. If we use a linked-list implementation instead, there won't be any preset limit on the size of a stack.

Our implementation will be similar to the one in the `stack2.c` file of Section 19.2. The linked list will consist of nodes, represented by the following structure:

```
struct node {
  Item data;
  struct node *next;
};
```

The type of the `data` member is now `Item` rather than `int`, but the structure is otherwise the same as before.

The `stack_type` structure will contain a pointer to the first node in the list:

```
struct stack_type {
  struct node *top;
};
```

At first glance, the `stack_type` structure seems superfluous; we could just define `Stack` to be `struct node *` and let a `Stack` value be a pointer to the first node in the list. However, we still need the `stack_type` structure so that the interface to the stack remains unchanged. (If we did away with it, any function that modified the stack would need a `Stack *` parameter instead of a `Stack` parameter.) Moreover, having the `stack_type` structure will make it easier to change the implementation in the future, should we decide to store additional information. For example, if we later decide that the `stack_type` structure should contain a count of how many items are currently stored in the stack, we can easily add a member to the `stack_type` structure to store this information.

We won't need to make any changes to the `stackADT.h` header. (We'll use this header file, not `stackADT2.h`.) We can also use the original `stackclient.c` file for testing. All the changes will be in the `stackADT.c` file. Here's the new version:

stackADT3.c

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

struct node {
  Item data;
  struct node *next;
};

struct stack_type {
  struct node *top;
};

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}
```

```
Stack create(void)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL)
    terminate("Error in create: stack could not be created.");
  s->top = NULL;
  return s;
}

void destroy(Stack s)
{
  make_empty(s);
  free(s);
}

void make_empty(Stack s)
{
  while (!is_empty(s))
    pop(s);
}

bool is_empty(Stack s)
{
  return s->top == NULL;
}

bool is_full(Stack s)
{
  return false;
}

void push(Stack s, Item i)
{
  struct node *new_node = malloc(sizeof(struct node));
  if (new_node == NULL)
    terminate("Error in push: stack is full.");

  new_node->data = i;
  new_node->next = s->top;
  s->top = new_node;
}

Item pop(Stack s)
{
  struct node *old_top;
  Item i;

  if (is_empty(s))
    terminate("Error in pop: stack is empty.");

  old_top = s->top;
  i = old_top->data;
  s->top = old_top->next;
  free(old_top);
  return i;
}
```

Note that the `destroy` function calls `make_empty` (to release the memory occupied by the nodes in the linked list) before it calls `free` (to release the memory for the `stack_type` structure).

19.5 Design Issues for Abstract Data Types

Section 19.4 described a stack ADT and showed several ways to implement it. Unfortunately, this ADT suffers from several problems that prevent it from being industrial-strength. Let's look at each of these problems and discuss possible solutions.

Naming Conventions

The stack ADT functions currently have short, easy-to-understand names: `create`, `destroy`, `make_empty`, `is_empty`, `is_full`, `push`, and `pop`. If we have more than one ADT in a program, name clashes are likely, with functions in two modules having the same name. (Each ADT will need its own `create` function, for example.) Therefore, we'll probably need to use function names that incorporate the name of the ADT itself, such as `stack_create` instead of `create`.

Error Handling

The stack ADT deals with errors by displaying an error message and terminating the program. That's not a bad thing to do. The programmer can avoid popping an empty stack or pushing data onto a full stack by being careful to call `is_empty` prior to each call of `pop` and `is_full` prior to each call of `push`, so in theory there's no reason for a call of `push` or `pop` to fail. (In the linked-list implementation, however, calling `is_full` isn't foolproof; a subsequent call of `push` can still fail.) Nevertheless, we might want to provide a way for a program to recover from these errors rather than terminating.

An alternative is to have the `push` and `pop` functions return a `bool` value to indicate whether or not they succeeded. `push` currently has a `void` return type, so it would be easy to modify it to return `true` if the `push` operation succeeds and `false` if the stack is full. Modifying the `pop` function would be more difficult, since `pop` currently returns the value that was popped. However, if `pop` were to return a *pointer* to this value, instead of the value itself, then `pop` could return `NULL` to indicate that the stack is empty.

A final comment about error handling: The C standard library contains a parameterized macro named `assert` that can terminate a program if a specified condition isn't satisfied. We could use calls of this macro as replacements for the `if` statements and calls of `terminate` that currently appear in the stack ADT.

`assert` macro ➤24.1

Generic ADTs

Midway through Section 19.4, we improved the stack ADT by making it easier to change the type of items stored in a stack—all we had to do was modify the definition of the `Item` type. It's still somewhat of a nuisance to do so; it would be nicer if a stack could accommodate items of any type, without the need to modify the `stack.h` file. Also note that our stack ADT suffers from a serious flaw: a program can't create two stacks whose items have different types. It's easy to create multiple stacks, but those stacks must have items with identical types. To allow stacks with different item types, we'd have to make copies of the stack ADT's header file and source file and modify one set of files so that the `Stack` type and its associated functions have different names.

What we'd like to have is a single "generic" stack type from which we could create a stack of integers, a stack of strings, or any other stack that we might need. There are various ways to create such a type in C, but none are completely satisfactory. The most common approach uses `void *` as the item type, which allows arbitrary pointers to be pushed and popped. With this technique, the `stack-ADT.h` file would be similar to our original version; however, the prototypes of the `push` and `pop` functions would have the following appearance:

```
void push(Stack s, void *p);
void *pop(Stack s);
```

`pop` returns a pointer to the item popped from the stack; if the stack is empty, it returns a null pointer.

There are two disadvantages to using `void *` as the item type. One is that this approach doesn't work for data that can't be represented in pointer form. Items could be strings (which are represented by a pointer to the first character in the string) or dynamically allocated structures but not basic types such as `int` and `double`. The other disadvantage is that error checking is no longer possible. A stack that stores `void *` items will happily allow a mixture of pointers of different types; there's no way to detect an error caused by pushing a pointer of the wrong type.

ADTs in Newer Languages

The problems that we've just discussed are dealt with much more cleanly in newer C-based languages, such as C++, Java, and C#. Name clashes are prevented by defining function names within a *class*. A stack ADT would be represented by a `Stack` class; the stack functions would belong to this class, and would only be recognized by the compiler when applied to a `Stack` object. These languages have a feature known as ***exception handling*** that allows functions such as `push` and `pop` to "throw" an exception when they detect an error condition. Code in the client can then deal with the error by "catching" the exception. C++, Java, and C# also provide special features for defining generic ADTs. In C++, for example, we would define a stack ***template***, leaving the item type unspecified.

Q & A

Q: You said that C wasn't designed for writing large programs. Isn't UNIX a large program? [p. 483]

A: Not at the time C was designed. In a 1978 paper, Ken Thompson estimated that the UNIX kernel was about 10,000 lines of C code (plus a small amount of assembler). Other components of UNIX were of comparable size; in another 1978 paper, Dennis Ritchie and colleagues put the size of the PDP-11 C compiler at 9660 lines. By today's standards, these are indeed small programs.

Q: Are there any abstract data types in the C library?

A: Technically there aren't, but a few come close, including the `FILE` type (defined in `<stdio.h>`). Before performing an operation on a file, we must declare a variable of type `FILE *`:

FILE type ➤22.1

```
FILE *fp;
```

The `fp` variable will then be passed to various file-handling functions.

Programmers are expected to treat `FILE` as an abstraction. It's not necessary to know what a `FILE` is in order to use the `FILE` type. Presumably `FILE` is a structure type, but the C standard doesn't even guarantee that. In fact, it's better not to know too much about how `FILE` values are stored, since the definition of the `FILE` type can (and often does) vary from one C compiler to another.

Of course, we can always look in the `stdio.h` file and see what a `FILE` is. Having done so, there's nothing to prevent us from writing code to access the internals of a `FILE`. For example, we might discover that `FILE` is a structure with a member named `bsize` (the file's buffer size):

```
typedef struct {
  ...
  int bsize;    /* buffer size */
  ...
} FILE;
```

Once we know about the `bsize` member, there's nothing to prevent us from accessing the buffer size for a particular file:

```
printf("Buffer size: %d\n", fp->bsize);
```

Doing so isn't a good idea, however, because other C compilers might store the buffer size under a different name, or keep track of it in some entirely different way. Changing the `bsize` member is an even worse idea:

```
fp->bsize = 1024;
```

Unless we know all the details about how files are stored, this is a dangerous thing to do. Even if we *do* know the details, they may change with a different compiler or the next release of the same compiler.

Q: What other incomplete types are there besides incomplete structure types? [p. 492]

A: One of the most common incomplete types occurs when an array is declared with no specified size:

```
extern int a[];
```

After this declaration (which we first encountered in Section 15.2), `a` has an incomplete type, because the compiler doesn't know `a`'s length. Presumably `a` is defined in another file within the program; that definition will supply the missing length. Another incomplete type occurs in declarations that specify no length for an array but provide an initializer:

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

In this example, the array `a` initially has an incomplete type, but the type is completed by the initializer.

Declaring a union tag without specifying the members of the union also creates an incomplete type. Flexible array members (a C99 feature) have an incomplete type. Finally, `void` is an incomplete type. The `void` type has the unusual property that it can never be completed, thus making it impossible to declare a variable of this type.

C99

flexible array members ➤ *17.9*

Q: What other restrictions are there on the use of incomplete types? [p. 492]

A: The `sizeof` operator can't be applied to an incomplete type (not surprisingly, since the size of an incomplete type is unknown). A member of a structure or union (other than a flexible array member) can't have an incomplete type. Similarly, the elements of an array can't have an incomplete type. Finally, a parameter in a function definition can't have an incomplete type (although this is allowed in a function *declaration*). The compiler "adjusts" each array parameter in a function definition so that it has a pointer type, thus preventing it from having an incomplete type.

Exercises

Section 19.1

1. A *queue* is similar to a stack, except that items are added at one end but removed from the other in a *FIFO* (first-in, first-out) fashion. Operations on a queue might include:

   Inserting an item at the end of the queue
   Removing an item from the beginning of the queue
   Returning the first item in the queue (without changing the queue)
   Returning the last item in the queue (without changing the queue)
   Testing whether the queue is empty

   Write an interface for a queue module in the form of a header file named `queue.h`.

Section 19.2

W 2. Modify the `stack2.c` file to use the `PUBLIC` and `PRIVATE` macros.

3. (a) Write an array based implementation of the queue module described in Exercise 1. Use three integers to keep track of the queue's status, with one integer storing the position of the first empty slot in the array (used when an item is inserted), the second storing the position of the next item to be removed, and the third storing the number of items in the queue. An insertion or removal that would cause either of the first two integers to be incremented past the end of the array should instead reset the variable to zero, thus causing it to "wrap around" to the beginning of the array.

   (b) Write a linked-list implementation of the queue module described in Exercise 1. Use two pointers, one pointing to the first node in the list and the other pointing to the last node. When an item is inserted into the queue, add it to the end of the list. When an item is removed from the queue, delete the first node in the list.

Section 19.3

W 4. (a) Write an implementation of the `Stack` type, assuming that `Stack` is a structure containing a fixed-length array.

   (b) Redo the `Stack` type, this time using a linked-list representation instead of an array. (Show both `stack.h` and `stack.c`.)

5. Modify the `queue.h` header of Exercise 1 so that it defines a `Queue` type, where `Queue` is a structure containing a fixed-length array (see Exercise 3(a)). Modify the functions in `queue.h` to take a `Queue *` parameter.

Section 19.4

6. (a) Add a `peek` function to `stackADT.c`. This function will have a parameter of type `Stack`. When called, it returns the top item on the stack but doesn't modify the stack.

   (b) Repeat part (a), modifying `stackADT2.c` this time.

   (c) Repeat part (a), modifying `stackADT3.c` this time.

7. Modify `stackADT2.c` so that a stack automatically doubles in size when it becomes full. Have the `push` function dynamically allocate a new array that's twice as large as the old one and then copy the stack contents from the old array to the new one. Be sure to have `push` deallocate the old array once the data has been copied.

Programming Projects

1. Modify Programming Project 1 from Chapter 10 so that it uses the stack ADT described in Section 19.4. You may use any of the implementations of the ADT described in that section.

2. Modify Programming Project 6 from Chapter 10 so that it uses the stack ADT described in Section 19.4. You may use any of the implementations of the ADT described in that section.

3. Modify the `stackADT3.c` file of Section 19.4 by adding an `int` member named `len` to the `stack_type` structure. This member will keep track of how many items are currently stored in a stack. Add a new function named `length` that has a `Stack` parameter and returns the value of the `len` member. (Some of the existing functions in `stackADT3.c` will need to be modified as well.) Modify `stackclient.c` so that it calls the `length` function (and displays the value that it returns) after each operation that modifies a stack.

4. Modify the `stackADT.h` and `stackADT3.c` files of Section 19.4 so that a stack stores values of type `void *`, as described in Section 19.5; the `Item` type will no longer be used. Modify `stackclient.c` so that it stores pointers to strings in the `s1` and `s2` stacks.

5. Starting from the `queue.h` header of Exercise 1, create a file named `queueADT.h` that defines the following `Queue` type:

   ```
   typedef struct queue_type *Queue;
   ```

   `queue_type` is an incomplete structure type. Create a file named `queueADT.c` that contains the full definition of `queue_type` as well as definitions for all the functions in `queue.h`. Use a fixed-length array to store the items in a queue (see Exercise 3(a)). Create a file named `queueclient.c` (similar to the `stackclient.c` file of Section 19.4) that creates two queues and performs operations on them. Be sure to provide `create` and `destroy` functions for your ADT.

6. Modify Programming Project 5 so that the items in a queue are stored in a dynamically allocated array whose length is passed to the `create` function.

7. Modify Programming Project 5 so that the items in a queue are stored in a linked list (see Exercise 3(b)).

20 Low-Level Programming

A programming language is low level when its programs require attention to the irrelevant.

Previous chapters have described C's high-level, machine-independent features. Although these features are adequate for many applications, some programs need to perform operations at the bit level. Bit manipulation and other low-level operations are especially useful for writing systems programs (including compilers and operating systems), encryption programs, graphics programs, and programs for which fast execution and/or efficient use of space is critical.

Section 20.1 covers C's bitwise operators, which provide easy access to both individual bits and bit-fields. Section 20.2 then shows how to declare structures that contain bit-fields. Finally, Section 20.3 describes how certain ordinary C features (type definitions, unions, and pointers) can help in writing low-level programs.

Some of the techniques described in this chapter depend on knowledge of how data is stored in memory, which can vary depending on the machine and the compiler. Relying on these techniques will most likely make a program nonportable, so it's best to avoid them unless absolutely necessary. If you do need them, try to limit their use to certain modules in your program; don't spread them around. And, above all, be sure to document what you're doing!

20.1 Bitwise Operators

C provides six ***bitwise operators***, which operate on integer data at the bit level. We'll discuss the two bitwise shift operators first, followed by the four other bitwise operators (bitwise complement, bitwise *and*, bitwise exclusive *or*, and bitwise inclusive *or*).

Bitwise Shift Operators

The bitwise shift operators can transform the binary representation of an integer by shifting its bits to the left or right. C provides two shift operators, which are shown in Table 20.1.

Table 20.1
Bitwise Shift Operators

Symbol	*Meaning*
<<	left shift
>>	right shift

The operands for << and >> may be of any integer type (including `char`). The integer promotions are performed on both operands: the result has the type of the left operand after promotion.

The value of `i << j` is the result when the bits in `i` are shifted left by `j` places. For each bit that is "shifted off" the left end of `i`, a zero bit enters at the right. The value of `i >> j` is the result when `i` is shifted right by `j` places. If `i` is of an unsigned type or if the value of `i` is nonnegative, zeros are added at the left as needed. If `i` is a negative number, the result is implementation-defined: some implementations add zeros at the left end, while others preserve the sign bit by adding ones.

portability tip *For portability, it's best to perform shifts only on unsigned numbers.*

The following examples illustrate the effect of applying the shift operators to the number 13. (For simplicity, these examples—and others in this section—use short integers, which are typically 16 bits.)

```
unsigned short i, j;

i = 13;         /* i is now 13 (binary 0000000000001101) */
j = i << 2;     /* j is now 52 (binary 0000000000110100) */
j = i >> 2;     /* j is now  3 (binary 0000000000000011) */
```

As these examples show, neither operator modifies its operands. To modify a variable by shifting its bits, we'd use the compound assignment operators <<= and >>=:

```
i = 13;         /* i is now 13 (binary 0000000000001101) */
i <<= 2;        /* i is now 52 (binary 0000000000110100) */
i >>= 2;        /* i is now 13 (binary 0000000000001101) */
```

⚠ The bitwise shift operators have lower precedence than the arithmetic operators, which can cause surprises. For example, `i << 2 + 1` means `i << (2 + 1)`, not `(i << 2) + 1`.

Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or*

Table 20.2 lists the remaining bitwise operators.

Table 20.2
Other Bitwise Operators

Symbol	*Meaning*
~	bitwise complement
&	bitwise *and*
^	bitwise exclusive *or*
|	bitwise inclusive *or*

The ~ operator is unary; the integer promotions are performed on its operand. The other operators are binary; the usual arithmetic conversions are performed on their operands.

The ~, &, ^, and | operators perform Boolean operations on all bits in their operands. The ~ operator produces the complement of its operand, with zeros replaced by ones and ones replaced by zeros. The & operator performs a Boolean *and* operation on all corresponding bits in its two operands. The ^ and | operators are similar (both perform a Boolean *or* operation on the bits in their operands); however, ^ produces 0 whenever both operands have a 1 bit, whereas | produces 1.

Don't confuse the *bitwise* operators & and | with the *logical* operators && and | |. The bitwise operators sometimes produce the same results as the logical operators, but they're not equivalent.

The following examples illustrate the effect of the ~, &, ^, and | operators:

```
unsigned short i, j, k;

i = 21;      /* i is now    21 (binary 0000000000010101) */
j = 56;      /* j is now    56 (binary 0000000000111000) */
k = ~i;      /* k is now 65514 (binary 1111111111101010) */
k = i & j;   /* k is now    16 (binary 0000000000010000) */
k = i ^ j;   /* k is now    45 (binary 0000000000101101) */
k = i | j;   /* k is now    61 (binary 0000000000111101) */
```

The value shown for `~i` is based on the assumption that an `unsigned short` value occupies 16 bits.

The ~ operator deserves special mention, since we can use it to help make even low-level programs more portable. Suppose that we need an integer whose bits are all 1. The preferred technique is to write `~0`, which doesn't depend on the number of bits in an integer. Similarly, if we need an integer whose bits are all 1 except for the last five, we could write `~0x1f`.

Each of the ~, &, ^, and | operators has a different precedence:

Highest: ~
&
^
Lowest: |

As a result, we can combine these operators in expressions without having to use parentheses. For example, we could write `i & ~j | k` instead of `(i & (~j)) | k` and `i ^ j & ~k` instead of `i ^ (j & (~k))`. Of course, it doesn't hurt to use parentheses to avoid confusion.

table of operators ➤ *Appendix A*

The precedence of &, ^, and | is lower than the precedence of the relational and equality operators. Consequently, statements like the following one won't have the desired effect:

```
if (status & 0x4000 != 0) ...
```

Instead of testing whether `status & 0x4000` isn't zero, this statement will evaluate `0x4000 != 0` (which has the value 1), then test whether the value of `status & 1` isn't zero.

The compound assignment operators &=, ^=, and |= correspond to the bitwise operators &, ^, and |:

```
i = 21;    /* i is now 21 (binary 0000000000010101) */
j = 56;    /* j is now 56 (binary 0000000000111000) */
i &= j;    /* i is now 16 (binary 0000000000010000) */
i ^= j;    /* i is now 40 (binary 0000000000101000) */
i |= j;    /* i is now 56 (binary 0000000000111000) */
```

Using the Bitwise Operators to Access Bits

When we do low-level programming, we'll often need to store information as single bits or collections of bits. In graphics programming, for example, we may want to squeeze two or more pixels into a single byte. Using the bitwise operators, we can extract or modify data that's stored in a small number of bits.

Let's assume that `i` is a 16-bit `unsigned short` variable. Let's see how to perform the most common single-bit operations on `i`:

- *Setting a bit.* Suppose that we want to set bit 4 of `i`. (We'll assume that the leftmost—or *most significant*—bit is numbered 15 and the least significant is numbered 0.) The easiest way to set bit 4 is to *or* the value of `i` with the constant `0x0010` (a "mask" that contains a 1 bit in position 4):

  ```
  i = 0x0000;          /* i is now 0000000000000000 */
  i |= 0x0010;         /* i is now 0000000000010000 */
  ```

  More generally, if the position of the bit is stored in the variable `j`, we can use a shift operator to create the mask:

idiom

```
i |= 1 << j;          /* sets bit j */
```

For example, if `j` has the value 3, then `1 << j` is `0x0008`.

- *Clearing a bit.* To clear bit 4 of `i`, we'd use a mask with a 0 bit in position 4 and 1 bits everywhere else:

```
i = 0x00ff;          /* i is now 0000000011111111 */
i &= ~0x0010;        /* i is now 0000000011101111 */
```

Using the same idea, we can easily write a statement that clears a bit whose position is stored in a variable:

idiom

```
i &= ~(1 << j);      /* clears bit j */
```

- *Testing a bit.* The following `if` statement tests whether bit 4 of `i` is set:

```
if (i & 0x0010) ...  /* tests bit 4 */
```

To test whether bit `j` is set, we'd use the following statement:

idiom

```
if (i & 1 << j) ...  /* tests bit j */
```

To make working with bits easier, we'll often give them names. For example, suppose that we want bits 0, 1, and 2 of a number to correspond to the colors blue, green, and red, respectively. First, we define names that represent the three bit positions:

```
#define BLUE  1
#define GREEN 2
#define RED   4
```

Setting, clearing, and testing the `BLUE` bit would be done as follows:

```
i |= BLUE;          /* sets BLUE bit   */
i &= ~BLUE;         /* clears BLUE bit */
if (i & BLUE) ...   /* tests BLUE bit  */
```

It's also easy to set, clear, or test several bits at time:

```
i |= BLUE | GREEN;           /* sets BLUE and GREEN bits   */
i &= ~(BLUE | GREEN);        /* clears BLUE and GREEN bits */
if (i & (BLUE | GREEN)) ...  /* tests BLUE and GREEN bits  */
```

The `if` statement tests whether either the `BLUE` bit *or* the `GREEN` bit is set.

Using the Bitwise Operators to Access Bit-Fields

Dealing with a group of several consecutive bits (a ***bit-field***) is slightly more complicated than working with single bits. Here are examples of the two most common bit-field operations:

- ***Modifying a bit-field.*** Modifying a bit-field requires a bitwise *and* (to clear the bit-field), followed by a bitwise *or* (to store new bits in the bit-field). The following statement shows how we might store the binary value 101 in bits 4–6 of the variable `i`:

```
i = i & ~0x0070 | 0x0050;   /* stores 101 in bits 4-6 */
```

The & operator clears bits 4–6 of `i`; the | operator then sets bits 6 and 4. Notice that `i |= 0x0050` by itself wouldn't always work: it would set bits 6 and 4 but not change bit 5. To generalize the example a little, let's assume that the variable `j` contains the value to be stored in bits 4–6 of `i`. We'll need to shift `j` into position before performing the bitwise *or*:

```
i = (i & ~0x0070) | (j << 4);   /* stores j in bits 4-6 */
```

The | operator has lower precedence than & and <<, so we can drop the parentheses if we wish:

```
i = i & ~0x0070 | j << 4;
```

- ***Retrieving a bit-field.*** When the bit-field is at the right end of a number (in the least significant bits), fetching its value is easy. For example, the following statement retrieves bits 0–2 in the variable `i`:

  ```
  j = i & 0x0007;              /* retrieves bits 0-2 */
  ```

  If the bit-field isn't at the right end of of `i`, then we can first shift the bit-field to the end before extracting the field using the & operator. To extract bits 4–6 of `i`, for example, we could use the following statement:

  ```
  j = (i >> 4) & 0x0007;   /* retrieves bits 4-6 */
  ```

PROGRAM XOR Encryption

One of the simplest ways to encrypt data is to exclusive-*or* (XOR) each character with a secret key. Suppose that the key is the & character. If we XOR this key with the character z, we'll get the \ character (assuming that we're using the ASCII character set):

ASCII character set ➤*Appendix E*

```
   00100110  (ASCII code for &)
XOR 01111010  (ASCII code for z)
   01011100  (ASCII code for \)
```

To decrypt a message, we just apply the same algorithm. In other words, by encrypting an already-encrypted message, we'll recover the original message. If we XOR the & character with the \ character, for example, we'll get the original character, z:

```
   00100110  (ASCII code for &)
XOR 01011100  (ASCII code for \)
   01111010  (ASCII code for z)
```

The following program, `xor.c`, encrypts a message by XORing each character with the & character. The original message can be entered by the user or read from a file using input redirection; the encrypted message can be viewed on the screen or saved in a file using output redirection. For example, suppose that the file

input and output redirection ➤*22.1*

msg contains the following lines:

```
Trust not him with your secrets, who, when left
alone in your room, turns over your papers.
                --Johann Kaspar Lavater (1741-1801)
```

To encrypt the msg file, saving the encrypted message in newmsg, we'd use the following command:

```
xor <msg >newmsg
```

newmsg will now contain these lines:

```
rTSUR HIR NOK QORN _IST UCETCRU, QNI, QNCH JC@R
GJIHC OH _IST TIIK, RSTHU IPCT _IST VGVCTU.
              --lINGHH mGUVGT jGPGRCT (1741-1801)
```

To recover the original message, we'd use the command

```
xor <newmsg
```

which will display it on the screen.

As the example shows, our program won't change some characters, including digits. XORing these characters with & would produce invisible control characters, which could cause problems with some operating systems. In Chapter 22, we'll see how to avoid problems when reading and writing files that contain control characters. Until then, we'll play it safe by using the `isprint` function to make sure that both the original character and the new (encrypted) character are printing characters (i.e., not control characters). If either character fails this test, we'll have the program write the original character instead of the new character.

`isprint` function ➤23.5

Here's the finished program, which is remarkably short:

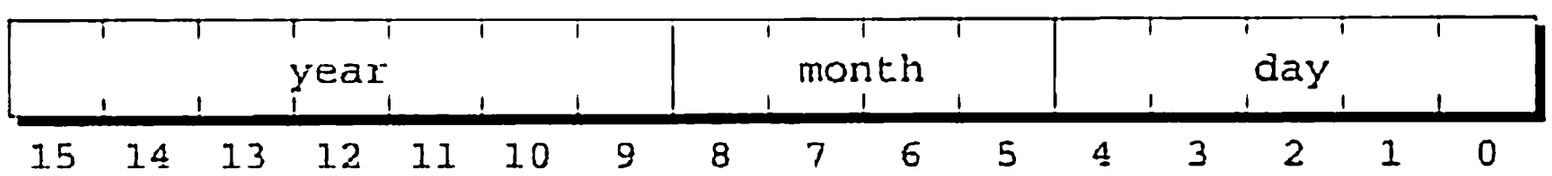
xor.c

```
/* Performs XOR encryption */

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

#define KEY '&'

int main(void)
{
  int orig_char, new_char;

  while ((orig_char = getchar()) != EOF) {
    new_char = orig_char ^ KEY;
    if (isprint(orig_char) && isprint(new_char))
      putchar(new_char);
    else
      putchar(orig_char);
  }

  return 0;
}
```

20.2 Bit-Fields in Structures

Although the techniques of Section 20.1 allow us to work with bit-fields, these techniques can be tricky to use and potentially confusing. Fortunately, C provides an alternative: declaring structures whose members represent bit-fields.

Q&A As an example, let's look at how the MS-DOS operating system (often just called DOS) stores the date at which a file was created or last modified. Since days, months, and years are small numbers, storing them as normal integers would waste space. Instead, DOS allocates only 16 bits for a date, with 5 bits for the day, 4 bits for the month, and 7 bits for the year:



Using bit-fields, we can define a C structure with an identical layout:

```
struct file_date {
  unsigned int day: 5;
  unsigned int month: 4;
  unsigned int year: 7;
};
```

The number after each member indicates its length in bits. Since the members all have the same type, we can condense the declaration if we want:

```
struct file_date {
  unsigned int day: 5, month: 4, year: 7;
};
```

The type of a bit-field must be either `int`, `unsigned int`, or `signed int`. Using `int` is ambiguous; some compilers treat the field's high-order bit as a sign bit, but others don't.

portability tip *Declare all bit-fields to be either `unsigned int` or `signed int`.*

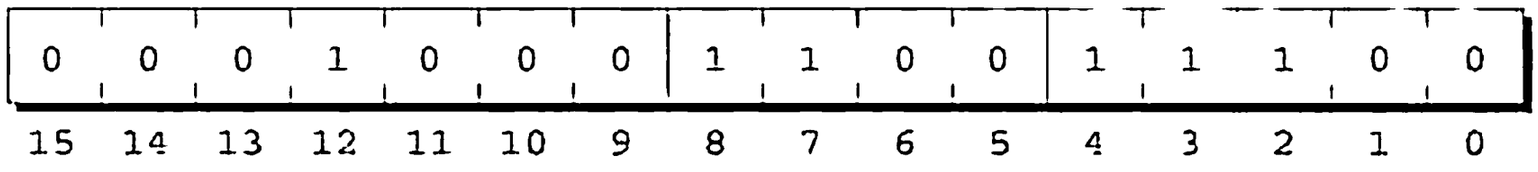
C99 In C99, bit-fields may also have type `_Bool`. C99 compilers may allow additional bit-field types.

We can use a bit-field just like any other member of a structure, as the following example shows:

```
struct file_date fd;

fd.day = 28;
fd.month = 12;
fd.year = 8;      /* represents 1988 */
```

Note that the `year` member is stored relative to 1980 (the year the world began.

according to Microsoft). After these assignments, the `fd` variable will have the following appearance:



We could have used the bitwise operators to accomplish the same effect; using these operators might even make the program a little faster. However, having a readable program is usually more important than gaining a few microseconds.

Bit-fields do have one restriction that doesn't apply to other members of a structure. Since bit-fields don't have addresses in the usual sense, C doesn't allow us to apply the address operator (`&`) to a bit-field. Because of this rule, functions such as `scanf` can't store data directly in a bit-field:

```
scanf("%d", &fd.day);    /*** WRONG ***/
```

Of course, we can always use `scanf` to read input into an ordinary variable and then assign it to `fd.day`.

How Bit-Fields Are Stored

Let's take a close look at how a compiler processes the declaration of a structure that has bit-field members. As we'll see, the C standard allows the compiler considerable latitude in choosing how it stores bit-fields.

The rules concerning how the compiler handles bit-fields depend on the notion of "storage units." The size of a storage unit is implementation-defined: typical values are 8 bits, 16 bits, and 32 bits. As it processes a structure declaration, the compiler packs bit-fields one by one into a storage unit, with no gaps between the fields, until there's not enough room for the next field. At that point, some compilers skip to the beginning of the next storage unit, while others split the bit-field across the storage units. (Which one occurs is implementation-defined.) The order in which bit-fields are allocated (left to right or right to left) is also implementation-defined.

Our `file_date` example assumes that storage units are 16 bits long. (An 8-bit storage unit would also be acceptable, provided that the compiler splits the `month` field across two storage units.) We also assume that bit-fields are allocated from right to left (with the first bit-field occupying the low-order bits).

C allows us to omit the name of any bit-field. Unnamed bit-fields are useful as "padding" to ensure that other bit fields are properly positioned. Consider the time associated with a DOS file, which is stored in the following way:

```
struct file_time {
  unsigned int seconds: 5;
  unsigned int minutes: 6;
  unsigned int hours: 5;
};
```

(You may be wondering how it's possible to store the seconds—a number between 0 and 59—in a field with only 5 bits. Well, DOS cheats: it divides the number of seconds by 2, so the `seconds` member is actually between 0 and 29.) If we're not interested in the `seconds` field, we can leave out its name:

```
struct file_time {
  unsigned int : 5;           /* not used */
  unsigned int minutes: 6;
  unsigned int hours: 5;
};
```

The remaining bit-fields will be aligned as if the `seconds` field were still present.

Another trick that we can use to control the storage of bit-fields is to specify 0 as the length of an unnamed bit-field:

```
struct s {
  unsigned int a: 4;
  unsigned int : 0;     /* 0-length bit-field */
  unsigned int b: 8;
};
```

A 0-length bit-field is a signal to the compiler to align the following bit-field at the beginning of a storage unit. If storage units are 8 bits long, the compiler will allocate 4 bits for the `a` member, skip 4 bits to the next storage unit, and then allocate 8 bits for `b`. If storage units are 16 bits long, the compiler will allocate 4 bits for `a`, skip 12 bits, and then allocate 8 bits for `b`.

20.3 Other Low-Level Techniques

Some of the language features that we've covered in previous chapters are used often in low-level programming. To wrap up this chapter, we'll take a look at several important examples: defining types that represent units of storage, using unions to bypass normal type-checking, and using pointers as addresses. We'll also cover the `volatile` type qualifier, which we avoided discussing in Section 18.3 because of its low-level nature.

Defining Machine-Dependent Types

Since the `char` type—by definition—occupies one byte, we'll sometimes treat characters as bytes, using them to store data that's not necessarily in character form. When we do so, it's a good idea to define a `BYTE` type:

```
typedef unsigned char BYTE;
```

Depending on the machine, we may want to define additional types. The x86 architecture makes extensive use of 16-bit words, so the following definition would be useful for that platform:

```
typedef unsigned short WORD;
```

We'll use the BYTE and WORD types in later examples.

Using Unions to Provide Multiple Views of Data

Although unions can be used in a portable way—see Section 16.4 for examples—they're often used in C for an entirely different purpose: viewing a block of memory in two or more different ways.

Here's a simple example based on the `file_date` structure described in Section 20.2. Since a `file_date` structure fits into two bytes, we can think of any two-byte value as a `file_date` structure. In particular, we could view an `unsigned short` value as a `file_date` structure (assuming that short integers are 16 bits long). The following union allows us to easily convert a short integer to a file date or vice versa:

```
union int_date {
  unsigned short i;
  struct file_date fd;
};
```

With the help of this union, we could fetch a file date from disk as two bytes, then extract its `month`, `day`, and `year` fields. Conversely, we could construct a date as a `file_date` structure, then write it to disk as a pair of bytes.

As an example of how we might use the `int_date` union, here's a function that, when passed an `unsigned short` argument, prints it as a file date:

```
void print_date(unsigned short n)
{
  union int_date u;

  u.i = n;
  printf("%d/%d/%d\n", u.fd.month, u.fd.day, u.fd.year + 1980);
}
```

Using unions to allow multiple views of data is especially useful when working with registers, which are often divided into smaller units. x86 processors, for example, have 16-bit registers named AX, BX, CX, and DX. Each of these registers can be treated as two 8-bit registers. AX, for example, is divided into registers named AH and AL. (The H and L stand for "high" and "low.")

When writing low-level applications for x86-based computers, we may need variables that represent the contents of the AX, BX, CX, and DX registers. We want access to both the 16- and 8-bit registers; at the same time, we need to take their relationships into account (a change to AX affects both AH and AL; changing AH or AL modifies AX). The solution is to set up two structures, one containing members that correspond to the 16-bit registers, and the other containing members that match the 8-bit registers. We then create a union that encloses the two structures:

```
union {
  struct {
    WORD ax, bx, cx, dx;
  } word;
  struct {
    BYTE al, ah, bl, bh, cl, ch, dl, dh;
  } byte;
} regs;
```

The members of the `word` structure will be overlaid with the members of the `byte` structure; for example, `ax` will occupy the same memory as `al` and `ah`. And that, of course, is exactly what we wanted. Here's an example showing how the `regs` union might be used:

```
regs.byte.ah = 0x12;
regs.byte.al = 0x34;
printf("AX: %hx\n", regs.word.ax);
```

Changing `ah` and `al` affects `ax`, so the output will be

```
AX: 1234
```

Note that the `byte` structure lists `al` before `ah`, even though the AL register is the "low" half of AX and AH is the "high" half. Here's the reason. When a data item consists of more than one byte, there are two logical ways to store it in memory: with the bytes in the "natural" order (with the leftmost byte stored first) or with the bytes in reverse order (the leftmost byte is stored last). The first alternative is called ***big-endian;*** the second is known as ***little-endian.*** C doesn't require a specific byte ordering, since that depends on the CPU on which a program will be executed. Some CPUs use the big-endian approach and some use the little-endian approach. What does this have to do with the `byte` structure? It turns out that x86 processors assume that data is stored in little-endian order, so the first byte of `regs.word.ax` is the low byte.

Q&A

We don't normally need to worry about byte ordering. However, programs that deal with memory at a low level must be aware of the order in which bytes are stored (as the `regs` example illustrates). It's also relevant when working with files that contain non-character data.

Be careful when using unions to provide multiple views of data. Data that is valid in its original format may be invalid when viewed as a different type, causing unexpected problems.

Using Pointers as Addresses

We saw in Section 11.1 that a pointer is really some kind of memory address, although we usually don't need to know the details. When we do low-level programming, however, the details matter.

An address often has the same number of bits as an integer (or long integer). Creating a pointer that represents a specific address is easy: we just cast an integer into a pointer. For example, here's how we might store the address 1000 (hex) in a pointer variable:

```
BYTE *p;

p = (BYTE *) 0x1000;    /* p contains address 0x1000 */
```

PROGRAM

Viewing Memory Locations

Our next program allows the user to view segments of computer memory; it relies on C's willingness to allow an integer to be used as a pointer. Most CPUs execute programs in "protected mode," however, which means that a program can access only those portions of memory that belong to the program. This prevents a program from accessing (or changing) memory that belongs to another application or to the operating system itself. As a result, we'll only be able to use our program to view areas of memory that have been allocated for use by the program itself. Going outside these regions will cause the program to crash.

The `viewmemory.c` program begins by displaying the address of its own `main` function as well as the address of one of its variables. This will give the user a clue as to which areas of memory can be probed. The program next prompts the user to enter an address (in the form of a hexadecimal integer) plus the number of bytes to view. The program then displays a block of bytes of the chosen length, starting at the specified address.

Bytes are displayed in groups of 10 (except for the last group, which may have fewer than 10 bytes). The address of a group of bytes is displayed at the beginning of a line, followed by the bytes in the group (displayed as hexadecimal numbers); followed by the same bytes displayed as characters (just in case the bytes happen to represent characters, as some of them may). Only printing characters (as determined by the `isprint` function) will be displayed; other characters will be shown as periods.

We'll assume that `int` values are stored using 32 bits and that addresses are also 32 bits long. Addresses are displayed in hexadecimal, as is customary.

viewmemory.c

```
/* Allows the user to view regions of computer memory */

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

typedef unsigned char BYTE;

int main(void)
{
  unsigned int addr;
  int i, n;
  BYTE *ptr;

  printf("Address of main function: %x\n", (unsigned int) main);
  printf("Address of addr variable: %x\n", (unsigned int) &addr);
```

```
  printf("\nEnter a (hex) address: ");
  scanf("%x", &addr);
  printf("Enter number of bytes to view: ");
  scanf("%d", &n);

  printf("\n");
  printf(" Address                Bytes                Characters\n");
  printf(" -------  ------------------------------  ----------\n");

  ptr = (BYTE *) addr;
  for (; n > 0; n -= 10) {
    printf("%8X  ", (unsigned int) ptr);
    for (i = 0; i < 10 && i < n; i++)
      printf("%.2X ", *(ptr + i));
    for (; i < 10; i++)
      printf("   ");
    printf(" ");
    for (i = 0;  i < 10 && i < n; i++) {
      BYTE ch = *(ptr + i);
      if (!isprint(ch))
        ch = '.';
      printf("%c", ch);
    }
    printf("\n");
    ptr += 10;
  }

  return 0;
}
```

The program is complicated somewhat by the possibility that the value of n isn't a multiple of 10, so there may be fewer than 10 bytes in the last group. Two of the `for` statements are controlled by the condition `i < 10 && i < n`. This condition causes the loops to execute 10 times or `n` times, whichever is smaller. There's also a `for` statement that compensates for any missing bytes in the last group by displaying three spaces for each missing byte. That way, the characters that follow the last group of bytes will align properly with the character groups on previous lines.

The `%X` conversion specifier used in this program is similar to `%x`, which was discussed in Section 7.1. The difference is that `%X` displays the hexadecimal digits A, B, C, D, E, and F as upper-case letters; `%x` displays them in lower case.

Here's what happened when I compiled the program using GCC and tested it on an x86 system running Linux:

```
Address of main function: 804847c
Address of addr variable: bff41154

Enter a (hex) address: 8048000
Enter number of bytes to view: 40

 Address                Bytes                Characters
 -------  ------------------------------  ----------
 8048000  7F 45 4C 46 01 01 01 00 00 00  .ELF......
 804800A  00 00 00 00 00 00 02 00 03 00  ..........
 8048014  01 00 00 00 C0 83 04 08 34 00  ........4.
 804801E  00 00 C0 0A 00 00 00 00 00 00  ..........
```

I asked the program to display 40 bytes starting at address 8048000, which precedes the address of the `main` function. Note the 7F byte followed by bytes representing the letters E, L, and F. These four bytes identify the format (ELF) in which the executable file was stored. ELF (Executable and Linking Format) is widely used by UNIX systems, including Linux. 8048000 is the default address at which ELF executables are loaded on x86 platforms.

Let's run the program again, this time displaying a block of bytes that starts at the address of the addr variable:

```
Address of main function: 804847c
Address of addr variable: bfec5484

Enter a (hex) address: bfec5484
Enter number of bytes to view: 64

 Address                 Bytes                    Characters
 -------  -------------------------------------  ----------
BFEC5484  84 54 EC BF B0 54 EC BF F4 6F  .T...T...o
BFEC548E  68 00 34 55 EC BF C0 54 EC BF  h.4U...T..
BFEC5498  08 55 EC BF E3 3D 57 00 00 00  .U...=W...
BFEC54A2  00 00 A0 BC 55 00 08 55 EC BF  ....U..U..
BFEC54AC  E3 3D 57 00 01 00 00 00 34 55  .=W.....4U
BFEC54B6  EC BF 3C 55 EC BF 56 11 55 00  ..<U..V.U.
BFEC54C0  F4 6F 68 00                    .oh.
```

None of the data stored in this region of memory is in character form, so it's a bit hard to follow. However, we do know one thing: the `addr` variable occupies the first four bytes of this region. When reversed, these bytes form the number BFEC5484, the address entered by the user. Why the reversal? Because x86 processors store data in little-endian order, as we saw earlier in this section.

The `volatile` Type Qualifier

On some computers, certain memory locations are "volatile"; the value stored at such a location can change as a program is running, even though the program itself isn't storing new values there. For example, some memory locations might hold data coming directly from input devices.

The `volatile` type qualifier allows us to inform the compiler if any of the data used in a program is volatile. `volatile` typically appears in the declaration of a pointer variable that will point to a volatile memory location:

```
volatile BYTE *p;   /* p will point to a volatile byte */
```

To see why `volatile` is needed, suppose that `p` points to a memory location that contains the most recent character typed at the user's keyboard. This location is volatile: its value changes each time the user enters a character. We might use the following loop to obtain characters from the keyboard and store them in a buffer array:

```
while (buffer not full) {
  wait for input;
  buffer[i] = *p;
  if (buffer[i++] == '\n')
    break;
}
```

A sophisticated compiler might notice that this loop changes neither `p` nor `*p`, so it could optimize the program by altering it so that `*p` is fetched just once:

```
store *p in a register;
while (buffer not full) {
  wait for input;
  buffer[i] = value stored in register;
  if (buffer[i++] == '\n')
    break;
}
```

The optimized program will fill the buffer with many copies of the same character—not exactly what we had in mind. Declaring that `p` points to volatile data avoids this problem by telling the compiler that `*p` must be fetched from memory each time it's needed.

Q & A

Q: What do you mean by saying that the `&` and `|` operators sometimes produce the same results as the `&&` and `||` operators, but not always? [p. 511]

A: Let's compare `i & j` with `i && j` (similar remarks apply to `|` and `||`). As long as `i` and `j` have the value 0 or 1 (in any combination), the two expressions will have the same value. However, if `i` and `j` should have other values, the expressions may not always match. If `i` is 1 and `j` is 2, for example, then `i & j` has the value 0 (`i` and `j` have no corresponding 1 bits), while `i && j` has the value 1. If `i` is 3 and `j` is 2, then `i & j` has the value 2, while `i && j` has the value 1.

Side effects are another difference. Evaluating `i & j++` *always* increments `j` as a side effect, whereas evaluating `i && j++` *sometimes* increments `j`.

Q: Who cares how DOS stores file dates? Isn't DOS dead? [p. 516]

A: For the most part, yes. However, there are still plenty of files created years ago whose dates are stored in the DOS format. In any event, DOS file dates are a good example of how bit-fields are used.

Q: Where do the terms "big-endian" and "little-endian" come from? [p. 520]

A: In Jonathan Swift's novel *Gulliver's Travels*, the fictional islands of Lilliput and Blefuscu are perpetually at odds over whether to open boiled eggs on the big end or the little end. The choice is arbitrary, of course, just like the order of bytes in a data item.

Exercises

Section 20.1

*1. Show the output produced by each of the following program fragments. Assume that `i`, `j`, and `k` are `unsigned short` variables.

```
(a) i = 8; j = 9;
    printf("%d", i >> 1 + j >> 1);
(b) i = 1;
    printf("%d", i & ~i);
(c) i = 2; j = 1; k = 0;
    printf("%d", ~i & j ^ k);
(d) i = 7; j = 8; k = 9;
    printf("%d", i ^ j & k);
```

W 2. Describe a simple way to "toggle" a bit (change it from 0 to 1 or from 1 to 0). Illustrate the technique by writing a statement that toggles bit 4 of the variable `i`.

*3. Explain what effect the following macro has on its arguments. You may assume that the arguments have the same type.

```
#define M(x,y)  ((x)^=(y),(y)^=(x),(x)^=(y))
```

W 4. In computer graphics, colors are often stored as three numbers, representing red, green, and blue intensities. Suppose that each number requires eight bits, and we'd like to store all three values in a single long integer. Write a macro named `MK_COLOR` with three parameters (the red, green, and blue intensities). `MK_COLOR` should return a `long` in which the last three bytes contain the red, green, and blue intensities, with the red value as the last byte and the green value as the next-to-last byte.

5. Write macros named `GET_RED`, `GET_GREEN`, and `GET_BLUE` that, when given a color as an argument (see Exercise 4), return its 8-bit red, green, and blue intensities.

W 6. (a) Use the bitwise operators to write the following function:

```
unsigned short swap_bytes(unsigned short i);
```

`swap_bytes` should return the number that results from swapping the two bytes in `i`. (Short integers occupy two bytes on most computers.) For example, if `i` has the value `0x1234` (00010010 00110100 in binary), then `swap_bytes` should return `0x3412` (00110100 00010010 in binary). Test your function by writing a program that reads a number in hexadecimal, then writes the number with its bytes swapped:

```
Enter a hexadecimal number (up to four digits): 1234
Number with bytes swapped: 3412
```

Hint: Use the `%hx` conversion to read and write the hex numbers.

(b) Condense the `swap_bytes` function so that its body is a single statement.

7. Write the following functions:

```
unsigned int rotate_left(unsigned int i, int n);
unsigned int rotate_right(unsigned int i, int n);
```

`rotate_left` should return the result of shifting the bits in `i` to the left by `n` places, with the bits that were "shifted off" moved to the right end of `i`. (For example, the call

`rotate_left(0x12345678, 4)` should return `0x23456781` if integers are 32 bits long.) `rotate_right` is similar, but it should "rotate" bits to the right instead of the left.

W 8. Let `f` be the following function:

```
unsigned int f(unsigned int i, int m, int n)
{
  return (i >> (m + 1 - n)) & ~(~0 << n);
}
```

(a) What is the value of `~(~0 << n)`?
(b) What does this function do?

9. (a) Write the following function:

```
int count_ones(unsigned char ch);
```

`count_ones` should return the number of 1 bits in `ch`.

(b) Write the function in part (a) without using a loop.

10. Write the following function:

```
unsigned int reverse_bits(unsigned int n);
```

`reverse_bits` should return an unsigned integer whose bits are the same as those in `n` but in reverse order.

11. Each of the following macros defines the position of a single bit within an integer:

```
#define SHIFT_BIT 1
#define CTRL_BIT  2
#define ALT_BIT   4
```

The following statement is supposed to test whether any of the three bits have been set, but it never displays the specified message. Explain why the statement doesn't work and show how to fix it. Assume that `key_code` is an `int` variable.

```
if (key_code & (SHIFT_BIT | CTRL_BIT | ALT_BIT) == 0)
  printf("No modifier keys pressed\n");
```

12. The following function supposedly combines two bytes to form an unsigned short integer. Explain why the function doesn't work and show how to fix it.

```
unsigned short create_short(unsigned char high_byte,
                            unsigned char low_byte)
{
  return high_byte << 8 + low_byte;
}
```

*13. If `n` is an `unsigned int` variable, what effect does the following statement have on the bits in `n`?

```
n &= n - 1;
```

Hint: Consider the effect on `n` if this statement is executed more than once.

Section 20.2

W 14. When stored according to the IEEE floating-point standard, a `float` value consists of a 1-bit sign (the leftmost—or most significant—bit), an 8-bit exponent, and a 23-bit fraction, in that order. Design a structure type that occupies 32 bits, with bit-field members corresponding to the sign, exponent, and fraction. Declare the bit-fields to have type `unsigned int`. Check the manual for your compiler to determine the order of the bit-fields.

*15. (a) Assume that the variable `s` has been declared as follows:

```
struct {
  int flag: 1;
} s;
```

With some compilers, executing the following statements causes 1 to be displayed, but with other compilers, the output is –1. Explain the reason for this behavior.

```
s.flag = 1;
printf("%d\n", s.flag);
```

(b) How can this problem be avoided?

Section 20.3

16. Starting with the 386 processor, x86 CPUs have 32-bit registers named EAX, EBX, ECX, and EDX. The second half (the least significant bits) of these registers is the same as AX, BX, CX, and DX, respectively. Modify the `regs` union so that it includes these registers as well as the older ones. Your union should be set up so that modifying EAX changes AX and modifying AX changes the second half of EAX. (The other new registers will work in a similar fashion.) You'll need to add some "dummy" members to the `word` and `byte` structures, corresponding to the other half of EAX, EBX, ECX, and EDX. Declare the type of the new registers to be `DWORD` (double word), which should be defined as `unsigned long`. Don't forget that the x86 architecture is little-endian.

Programming Projects

1. Design a union that makes it possible to view a 32-bit value as either a `float` or the structure described in Exercise 14. Write a program that stores 1 in the structure's sign field, 128 in the exponent field, and 0 in the fraction field, then prints the `float` value stored in the union. (The answer should be –2.0 if you've set up the bit-fields correctly.)

21 The Standard Library

Every program is a part of some other program and rarely fits.

In previous chapters we've looked at the C library piecemeal; this chapter focuses on the library as a whole. Section 21.1 lists general guidelines for using the library. It also describes a trick found in some library headers: using a macro to "hide" a function. Section 21.2 gives an overview of each header in the C89 library; Section 21.3 does the same for the new headers in the C99 library.

Later chapters cover the library's headers in depth, with related headers grouped together into chapters. The `<stddef.h>` and `<stdbool.h>` headers are very brief, so I've chosen to discuss them in this chapter (in Sections 21.4 and 21.5, respectively).

21.1 Using the Library

C99

The C89 standard library is divided into 15 parts, with each part described by a header. C99 has an additional nine headers, for a total of 24 (see Table 21.1).

Table 21.1
Standard Library Headers

`<assert.h>`	`<inttypes.h>`†	`<signal.h>`	`<stdlib.h>`
`<complex.h>`†	`<iso646.h>`†	`<stdarg.h>`	`<string.h>`
`<ctype.h>`	`<limits.h>`	`<stdbool.h>`†	`<tgmath.h>`†
`<errno.h>`	`<locale.h>`	`<stddef.h>`	`<time.h>`
`<fenv.h>`†	`<math.h>`	`<stdint.h>`†	`<wchar.h>`†
`<float.h>`	`<setjmp.h>`	`<stdio.h>`	`<wctype.h>`†

†C99 only

Most compilers come with a more extensive library that invariably has many headers that don't appear in Table 21.1. The extra headers aren't standard, of

course, so we can't count on them to be available with other compilers. These headers often provide functions that are specific on a particular computer or operating system (which explains why they're not standard). They may provide functions that allow more control over the screen and keyboard. Headers that support graphics or a window-based user interface are also common.

The standard headers consist primarily of function prototypes, type definitions, and macro definitions. If one of our files contains a call of a function declared in a header or uses one of the types or macros defined there, we'll need to include the header at the beginning of the file. When a file includes several standard headers, the order of `#include` directives doesn't matter. It's also legal to include a standard header more than once.

Restrictions on Names Used in the Library

Any file that includes a standard header must obey a couple of rules. First, it can't use the names of macros defined in that header for any other purpose. If a file includes `<stdio.h>`, for example, it can't reuse `NULL`, since a macro by that name is already defined in `<stdio.h>`. Second, library names with file scope (`typedef` names, in particular) can't be redefined at the file level. Thus, if a file includes `<stdio.h>`, it can't define `size_t` as a identifier with file scope, since `<stdio.h>` defines `size_t` to be a `typedef` name.

Although these restrictions are pretty obvious, C has other restrictions that you might not expect:

- ***Identifiers that begin with an underscore followed by an upper-case letter or a second underscore*** are reserved for use within the library; programs should never use names of this form for any purpose.
- ***Identifiers that begin with an underscore*** are reserved for use as identifiers and tags with file scope. You should never use such a name for your own purposes unless it's declared inside a function.
- ***Every identifier with external linkage in the standard library*** is reserved for use as an identifier with external linkage. In particular, the names of all standard library functions are reserved. Thus, even if a file *doesn't* include `<stdio.h>`, it shouldn't define an external function named `printf`, since there's already a function with this name in the library.

These rules apply to *every* file in a program, regardless of which headers the file includes. Although these rules aren't always enforced, failing to obey them can lead to a program that's not portable.

The rules listed above apply not just to names that are currently used in the library, but also to names that are set aside for future use. The complete description of which names are reserved is rather lengthy; you'll find it in the C standard under "future library directions." As an example, C reserves identifiers that begin with `str` followed by a lower-case letter, so that functions with such names can be added to the `<string.h>` header.

Functions Hidden by Macros

Q&A

It's common for C programmers to replace small functions by parameterized macros. This practice occurs even in the standard library. The C standard allows headers to define macros that have the same names as library functions, but protects the programmer by requiring that a true function be available as well. As a result, it's not unusual for a library header to declare a function *and* define a macro with the same name.

We've already seen an example of a macro duplicating a library function. `getchar` is a library function declared in the `<stdio.h>` header. It has the following prototype:

```
int getchar(void);
```

`<stdio.h>` usually defines `getchar` as a macro as well:

```
#define getchar() getc(stdin)
```

By default, a call of `getchar` will be treated as a macro invocation (since macro names are replaced during preprocessing).

Most of the time, we're happy using a macro instead of a true function, because it will probably make our program run faster. Occasionally, though, we want a genuine function, perhaps to minimize the size of the executable code.

#undef directive ➤ 14.3

If the need arises, we can remove a macro definition (thus gaining access to the true function) by using the `#undef` directive. For example, we could undefine the `getchar` macro after including `<stdio.h>`:

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

If `getchar` *isn't* a macro, no harm has been done; `#undef` has no effect when given a name that's not defined as a macro.

As an alternative, we can disable individual uses of a macro by putting parentheses around its name:

```
ch = (getchar)();    /* instead of ch = getchar(); */
```

The preprocessor can't spot a parameterized macro unless its name is followed by a left parenthesis. The compiler isn't so easily fooled, however; it can still recognize `getchar` as a function.

21.2 C89 Library Overview

We'll now take a quick look at the headers in the C89 standard library. This section can serve as a "road map" to help you determine which part of the library you need. Each header is described in detail later in this chapter or in a subsequent chapter.

<assert.h> Diagnostics

<assert.h> header ➤24.1

Contains only the `assert` macro, which allows us to insert self-checks into a program. If any check fails, the program terminates.

<ctype.h> Character Handling

<ctype.h> header ➤23.5

Provides functions for classifying characters and for converting letters from lower to upper case or vice versa.

<errno.h> Errors

<errno.h> header ➤24.2

Provides `errno` ("error number"), an lvalue that can be tested after a call of certain library functions to see if an error occurred during the call.

<float.h> Characteristics of Floating Types

<float.h> header ➤23.1

Provides macros that describe the characteristics of floating types, including their range and accuracy.

<limits.h> Sizes of Integer Types

<limits.h> header ➤23.2

Provides macros that describe the characteristics of integer types (including character types), including their maximum and minimum values.

<locale.h> Localization

<locale.h> header ➤25.1

Provides functions to help a program adapt its behavior to a country or other geographic region. Locale-specific behavior includes the way numbers are printed (such as the character used as the decimal point), the format of monetary values (the currency symbol, for example), the character set, and the appearance of the date and time.

<math.h> Mathematics

<math.h> header ➤23.3

Provides common mathematical functions, including trigonometric, hyperbolic, exponential, logarithmic, power, nearest integer, absolute value, and remainder functions.

<setjmp.h> Nonlocal Jumps

<setjmp.h> header ➤24.4

Provides the `setjmp` and `longjmp` functions. `setjmp` "marks" a place in a program; `longjmp` can then be used to return to that place later. These functions

make it possible to jump from one function into another, still-active function, bypassing the normal function-return mechanism. `setjmp` and `longjmp` are used primarily for handling serious problems that arise during program execution.

`<signal.h>` *Signal Handling*

<signal.h> header ➤24.3

Provides functions that deal with exceptional conditions (signals), including interrupts and run-time errors. The `signal` function installs a function to be called if a given signal should occur later. The `raise` function causes a signal to occur.

`<stdarg.h>` *Variable Arguments*

<stdarg.h> header ➤26.1

Provides tools for writing functions that, like `printf` and `scanf`, can have a variable number of arguments.

`<stddef.h>` *Common Definitions*

<stddef.h> header ➤21.4

Provides definitions of frequently used types and macros.

`<stdio.h>` *Input/Output*

<stdio.h> header ➤22.1–22.8

Provides a large assortment of input/output functions, including operations on both sequential and random-access files.

`<stdlib.h>` *General Utilities*

<stdlib.h> header ➤26.2

A "catchall" header for functions that don't fit into any of the other headers. The functions in this header can convert strings to numbers, generate pseudo-random numbers, perform memory management tasks, communicate with the operating system, do searching and sorting, and perform conversions between multibyte characters and wide characters.

`<string.h>` *String Handling*

<string.h> header ➤23.6

Provides functions that perform string operations, including copying, concatenation, comparison, and searching, as well as functions that operate on arbitrary blocks of memory.

`<time.h>` *Date and Time*

<time.h> header ➤26.3

Provides functions for determining the time (and date), manipulating times, and formatting times for display.

21.3 C99 Library Changes

Some of the biggest changes in C99 affect the standard library. These changes fall into three groups:

- *Additional headers.* The C99 standard library has nine headers that don't exist in C89. Three of these (`<iso646.h>`, `<wchar.h>`, and `<wctype.h>`) were actually added to C in 1995 when the C89 standard was amended. The other six (`<complex.h>`, `<fenv.h>`, `<inttypes.h>`, `<stdbool.h>`, `<stdint.h>`, and `<tgmath.h>`) are new in C99.
- *Additional macros and functions.* The C99 standard adds macros and functions to several existing headers, primarily `<float.h>`, `<math.h>`, and `<stdio.h>`. The additions to the `<math.h>` header are so extensive that they're covered in a separate section (Section 23.4).
- *Enhanced versions of existing functions.* Some existing functions, including `printf` and `scanf`, have additional capabilities in C99.

We'll now take a quick look at the nine additional headers in the C99 standard library, just as we did in Section 21.2 for the headers in the C89 library.

`<complex.h>` *Complex Arithmetic*

<complex.h> header ➤27.4

Defines the `complex` and `I` macros, which are useful when working with complex numbers. Also provides functions for performing mathematical operations on complex numbers.

`<fenv.h>` *Floating-Point Environment*

<fenv.h> header ➤27.6

Provides access to floating-point status flags and control modes. For example, a program might test a flag to see if overflow occurred during a floating-point operation or set a control mode to specify how rounding should be done.

`<inttypes.h>` *Format Conversion of Integer Types*

<inttypes.h> header ➤27.2

Defines macros that can be used in format strings for input/output of the integer types declared in `<stdint.h>`. Also provides functions for working with greatest-width integers.

`<iso646.h>` *Alternative Spellings*

<iso646.h> header ➤25.3

Defines macros that represent certain operators (the ones containing the characters &, |, ~, !, and ^). These macros are useful for writing programs in an environment where these characters might not be part of the local character set.

<stdbool.h> Boolean Type and Values

<stdbool.h> header ➤21.5

Defines the `bool`, `true`, and `false` macros, as well as a macro that can be used to test whether these macros have been defined.

<stdint.h> Integer Types

<stdint.h> header ➤27.1

Declares integer types with specified widths and defines related macros (such as macros that specify the maximum and minimum values of each type). Also defines parameterized macros that construct integer constants with specific types.

<tgmath.h> Type-Generic Math

<tgmath.h> header ➤27.5

In C99, there are multiple versions of many math functions in the <math.h> and <complex.h> headers. The "type-generic" macros in <tgmath.h> can detect the types of the arguments passed to them and substitute a call of the appropriate <math.h> or <complex.h> function.

<wchar.h> Extended Multibyte and Wide-Character Utilities

<wchar.h> header ➤25.5

Provides functions for wide-character input/output and wide string manipulation.

<wctype.h> Wide-Character Classification and Mapping Utilities

<wctype.h> header ➤25.6

The wide-character version of <ctype.h>. Provides functions for classifying and changing the case of wide characters.

21.4 The <stddef.h> Header: Common Definitions

The <stddef.h> header provides definitions of frequently used types and macros; it doesn't declare any functions. The types are:

- `ptrdiff_t`. The type of the result when two pointers are subtracted.
- `size_t`. The type returned by the `sizeof` operator.
- `wchar_t`. A type large enough to represent all possible characters in all supported locales.

All three are names for integer types: `ptrdiff_t` must be a signed type, while `size_t` must be an unsigned type. For more information about `wchar_t`, see Section 25.2.

The <stddef.h> header also defines two macros. One of them is NULL, which represents the null pointer. The other macro, `offsetof`, requires two arguments: *type* (a structure type) and *member-designator* (a member of the structure).

`offsetof` computes the number of bytes between the beginning of the structure and the specified member.

Consider the following structure:

```
struct s {
  char a;
  int b[2];
  float c;
};
```

The value of `offsetof(struct s, a)` must be 0: C guarantees that the first member of a structure has the same address as the structure itself. We can't say for sure what the offsets of `b` and `c` are. One possibility is that `offsetof(struct s, b)` is 1 (since `a` is one byte long), and `offsetof(struct s, c)` is 9 (assuming 32-bit integers). However, some compilers leave "holes"—unused bytes—in structures (see the Q&A section at the end of Chapter 16), which can affect the value produced by `offsetof`. If a compiler should leave a three-byte hole after `a`, for example, then the offsets of `b` and `c` would be 4 and 12, respectively. But that's the beauty of `offsetof`: it produces the correct offsets for any compiler, enabling us to write portable programs.

There are various uses for `offsetof`. For example, suppose that we want to save the first two members of an `s` structure in a file, ignoring the `c` member. Instead of having the `fwrite` function write `sizeof(struct s)` bytes, which would save the entire structure, we'll tell it to write only `offsetof(struct s, c)` bytes.

fwrite function ➤22.6

A final remark: Some of the types and macros defined in `<stddef.h>` appear in other headers as well. (The `NULL` macro, for example, is also defined in `<locale.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, and `<time.h>`, as well as in the C99 header `<wchar.h>`.) As a result, few programs need to include `<stddef.h>`.

21.5 The `<stdbool.h>` Header (C99): Boolean Type and Values

The `<stdbool.h>` header defines four macros:

- `bool` (defined to be `_Bool`)
- `true` (defined to be `1`)
- `false` (defined to be `0`)
- `__bool_true_false_are_defined` (defined to be `1`)

We've seen many examples of how `bool`, `true`, and `false` are used. Potential uses of the `__bool_true_false_are_defined` macro are more limited. A program could use a preprocessing directive (such as `#if` or `#ifdef`) to test this macro before attempting to define its own version of `bool`, `true`, or `false`.

Q & A

Q: I notice that you use the term "standard header" rather than "standard header file." Is there any reason for not using the word "file"?

A: Yes. According to the C standard, a "standard header" need not be a file. Although most compilers do indeed store standard headers as files, the headers could in fact be built into the compiler itself.

Q: Section 14.3 described some disadvantages of using parameterized macros in place of functions. In light of these problems, isn't it dangerous to provide a macro substitute for a standard library function? [p. 531]

A: According to the C standard, a parameterized macro that substitutes for a library function must be "fully protected" by parentheses and must evaluate its arguments exactly once. These rules avoid most of the problems mentioned in Section 14.3.

Exercises

Section 21.1

1. Locate where header files are kept on your system. Find the nonstandard headers and determine the purpose of each.

2. Having located the header files on your system (see Exercise 1), find a standard header in which a macro hides a function.

3. When a macro hides a function, which must come first in the header file: the macro definition or the function prototype? Justify your answer.

4. Make a list of all reserved identifiers in the "future library directions" section of the C99 standard. Distinguish between identifiers that are reserved for use only when a specific header is included versus identifiers that are reserved for use as external names.

*5. The `islower` function, which belongs to `<ctype.h>`, tests whether a character is a lower-case letter. Why would the following macro version of `islower` not be legal, according to the C standard? (You may assume that the character set is ASCII.)

```
#define islower(c) ((c) >= 'a' && (c) <= 'z')
```

6. The `<ctype.h>` header usually defines most of its functions as macros as well. These macros rely on a static array that's declared in `<ctype.h>` but defined in a separate file. A portion of a typical `<ctype.h>` header appears below. Use this sample to answer the following questions.

   (a) Why do the names of the "bit" macros (such as `_UPPER`) and the `_ctype` array begin with an underscore?

   (b) Explain what the `_ctype` array will contain. Assuming that the character set is ASCII, show the values of the array elements at positions 9 (the horizontal tab character), 32 (the space character), 65 (the letter `A`), and 94 (the `^` character). See Section 23.5 for a description of what each macro should return.

(c) What's the advantage of using an array to implement these macros?

```
#define _UPPER    0x01    /* upper-case letter */
#define _LOWER    0x02    /* lower-case letter */
#define _DIGIT    0x04    /* decimal digit */
#define _CONTROL  0x08    /* control character */
#define _PUNCT    0x10    /* punctuation character */
#define _SPACE    0x20    /* white-space character */
#define _HEX      0x40    /* hexadecimal digit */
#define _BLANK    0x80    /* space character */

#define isalnum(c)   (_ctype[c] & (_UPPER|_LOWER|_DIGIT))
#define isalpha(c)   (_ctype[c] & (_UPPER|_LOWER))
#define iscntrl(c)   (_ctype[c] & _CONTROL)
#define isdigit(c)   (_ctype[c] & _DIGIT)
#define isgraph(c)   (_ctype[c] &
                        (_PUNCT|_UPPER|_LOWER|_DIGIT))
#define islower(c)   (_ctype[c] & _LOWER)
#define isprint(c)   ( ctype[c] &
                        (_BLANK|_PUNCT|_UPPER|_LOWER|_DIGIT))
#define ispunct(c)   (_ctype[c] & _PUNCT)
#define isspace(c)   (_ctype[c] & _SPACE)
#define isupper(c)   (_ctype[c] & _UPPER)
#define isxdigit(c)  (_ctype[c] & (_DIGIT|_HEX))
```

Section 21.2

W 7. In which standard header would you expect to find each of the following?

(a) A function that determines the current day of the week
(b) A function that tests whether a character is a digit
(c) A macro that gives the largest `unsigned int` value
(d) A function that rounds a floating-point number to the next higher integer
(e) A macro that specifies the number of bits in a character
(f) A macro that specifies the number of significant digits in a `double` value
(g) A function that searches a string for a particular character
(h) A function that opens a file for reading

Programming Projects

1. Write a program that declares the `s` structure (see Section 21.4) and prints the sizes and offsets of the `a`, `b`, and `c` members. (Use `sizeof` to find sizes; use `offsetof` to find offsets.) Have the program print the size of the entire structure as well. From this information, determine whether or not the structure has any holes. If it does, describe the location and size of each.

22 Input/Output

In man-machine symbiosis, it is man who must adjust: The machines can't.

C's input/output library is the biggest and most important part of the standard library. As befits its lofty status, we'll devote an entire chapter (the longest in the book) to the `<stdio.h>` header, the primary repository of input/output functions.

We've been using `<stdio.h>` since Chapter 2, and we have experience with the `printf`, `scanf`, `putchar`, `getchar`, `puts`, and `gets` functions. This chapter provides more information about these six functions, as well as introducing a host of new functions, most of which deal with files. Fortunately, many of the new functions are closely related to functions with which we're already acquainted. `fprintf`, for instance, is the "file version" of the `printf` function.

We'll start the chapter with a discussion of some basic issues: the stream concept, the `FILE` type, input and output redirection, and the difference between text files and binary files (Section 22.1). We'll then turn to functions that are designed specifically for use with files, including functions that open and close files (Section 22.2). After covering `printf`, `scanf`, and related functions for "formatted" input/output (Section 22.3), we'll look at functions that read and write unformatted data:

- `getc`, `putc`, and related functions, which read and write one *character* at a time (Section 22.4).
- `gets`, `puts`, and related functions, which read and write one *line* at a time (Section 22.5).
- `fread` and `fwrite`, which read and write *blocks* of data (Section 22.6).

Section 22.7 then shows how to perform random access operations on files. Finally, Section 22.8 describes the `sprintf`, `snprintf`, and `sscanf` functions, variants of `printf` and `scanf` that write to a string or read from a string.

This chapter covers all but eight of the functions in `<stdio.h>`. One of these eight, the `perror` function, is closely related to the `<errno.h>` header, so

I'll postpone it until Section 24.2, which discusses that header. Section 26.1 covers the remaining functions (`vfprintf`, `vprintf`, `vsprintf`, `vsnprintf`, `vfscanf`, `vscanf`, and `vsscanf`). These functions rely on the `va_list` type, which is introduced in that section.

C99

<wchar.h> header ➤25.5

In C89, all standard input/output functions belong to `<stdio.h>`, but such is not the case in C99, where some I/O functions are declared in the `<wchar.h>` header. The `<wchar.h>` functions deal with wide characters rather than ordinary characters; the good news is that most of these functions closely resemble those of `<stdio.h>`. Functions in `<stdio.h>` that read or write data are known as ***byte input/output functions;*** similar functions in `<wchar.h>` are called ***wide-character input/output functions.***

22.1 Streams

In C, the term *stream* means any source of input or any destination for output. Many small programs, like the ones in previous chapters, obtain all their input from one stream (usually associated with the keyboard) and write all their output to another stream (usually associated with the screen).

Larger programs may need additional streams. These streams often represent files stored on various media (such as hard drives, CDs, DVDs, and flash memory), but they could just as easily be associated with devices that don't store files: network ports, printers, and the like. We'll concentrate on files, since they're common and easy to understand. (I may even occasionally use the term *file* when I should say *stream*.) Keep in mind, however, that many of the functions in `<stdio.h>` work equally well with all streams, not just the ones that represent files.

File Pointers

Accessing a stream in a C program is done through a ***file pointer***, which has type `FILE *` (the `FILE` type is declared in `<stdio.h>`). Certain streams are represented by file pointers with standard names; we can declare additional file pointers as needed. For example, if a program needs two streams in addition to the standard ones, it might contain the following declaration:

```
FILE *fp1, *fp2;
```

A program may declare any number of `FILE *` variables, although operating systems usually limit the number of streams that can be open at one time.

Standard Streams and Redirection

`<stdio.h>` provides three standard streams (Table 22.1). These streams are ready to use—we don't declare them, and we don't open or close them.

Table 22.1
Standard Streams

File Pointer	*Stream*	*Default Meaning*
`stdin`	Standard input	Keyboard
`stdout`	Standard output	Screen
`stderr`	Standard error	Screen

The functions that we've used in previous chapters—`printf`, `scanf`, `putchar`, `getchar`, `puts`, and `gets`—obtain input from `stdin` and send output to `stdout`. By default, `stdin` represents the keyboard; `stdout` and `stderr` represent the screen. However, many operating systems allow these default meanings to be changed via a mechanism known as ***redirection***.

Q&A

Typically, we can force a program to obtain its input from a file instead of from the keyboard by putting the name of the file on the command line, preceded by the < character:

```
demo <in.dat
```

This technique, known as ***input redirection***, essentially makes the `stdin` stream represent a file (`in.dat`, in this case) instead of the keyboard. The beauty of redirection is that the `demo` program doesn't realize that it's reading from `in.dat`; as far as it knows, any data it obtains from `stdin` is being entered at the keyboard.

Output redirection is similar. Redirecting the `stdout` stream is usually done by putting a file name on the command line, preceded by the > character:

```
demo >out.dat
```

All data written to `stdout` will now go into the `out.dat` file instead of appearing on the screen. Incidentally, we can combine output redirection with input redirection:

```
demo <in.dat >out.dat
```

The < and > characters don't have to be adjacent to file names, and the order in which the redirected files are listed doesn't matter, so the following examples would work just as well:

```
demo < in.dat > out.dat
demo >out.dat <in.dat
```

One problem with output redirection is that *everything* written to `stdout` is put into a file. If the program goes off the rails and begins writing error messages, we won't see them until we look at the file. This is where `stderr` comes in. By writing error messages to `stderr` instead of `stdout`, we can guarantee that those messages will appear on the screen even when `stdout` has been redirected. (Operating systems often allow `stderr` itself to be redirected, though.)

Text Files versus Binary Files

`<stdio.h>` supports two kinds of files: text and binary. The bytes in a ***text file*** represent characters, making it possible for a human to examine the file or edit it.

The source code for a C program is stored in a text file, for example. In a ***binary file***, on the other hand, bytes don't necessarily represent characters; groups of bytes might represent other types of data, such as integers and floating-point numbers. An executable C program is stored in a binary file, as you'll quickly realize if you try to look at the contents of one.

Text files have two characteristics that binary files don't possess:

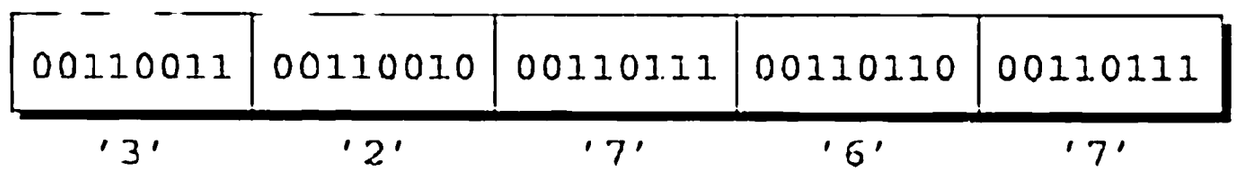
- ***Text files are divided into lines.*** Each line in a text file normally ends with one or two special characters; the choice of characters depends on the operating system. In Windows, the end-of-line marker is a carriage-return character (`'\x0d'`) followed immediately by a line-feed character (`'\x0a'`). In UNIX and newer versions of the Macintosh operating system (Mac OS), the end-of-line marker is a single line-feed character. Older versions of Mac OS use a single carriage-return character.

Q&A

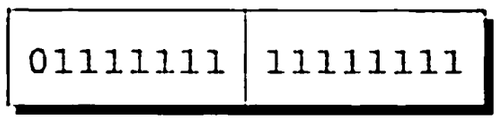
- ***Text files may contain a special "end-of-file" marker.*** Some operating systems allow a special byte to be used as a marker at the end of a text file. In Windows, the marker is `'\x1a'` (Ctrl-Z). There's no requirement that Ctrl-Z be present, but if it is, it marks the end of the file; any bytes after Ctrl-Z are to be ignored. The Ctrl-Z convention is a holdover from DOS, which in turn inherited it from CP/M, an early operating system for personal computers. Most other operating systems, including UNIX, have no special end-of-file character.

Binary files aren't divided into lines. In a binary file, there are no end-of-line or end-of-file markers; all bytes are treated equally.

Q&A When we write data to a file, we'll need to consider whether to store it in text form or in binary form. To see the difference, consider how we might store the number 32767 in a file. One option would be to write the number in text form as the characters 3, 2, 7, 6, and 7. If the character set is ASCII, we'd have the following five bytes:



The other option is to store the number in binary, which would take as few as two bytes:



little-endian order ➤*20.3* (The bytes will be reversed on systems that store data in little-endian order.) As this example shows, storing numbers in binary can often save quite a bit of space.

When we're writing a program that reads from a file or writes to a file, we need to take into account whether it's a text file or a binary file. A program that displays the contents of a file on the screen will probably assume it's a text file. A file-

copying program, on the other hand, can't assume that the file to be copied is a text file. If it does, binary files containing an end-of-file character won't be copied completely. When we can't say for sure whether a file is text or binary, it's safer to assume that it's binary.

22.2 File Operations

Simplicity is one of the attractions of input and output redirection; there's no need to open a file, close a file, or perform any other explicit file operations. Unfortunately, redirection is too limited for many applications. When a program relies on redirection, it has no control over its files; it doesn't even know their names. Worse still, redirection doesn't help if the program needs to read from two files or write to two files at the same time.

When redirection isn't enough, we'll end up using the file operations that `<stdio.h>` provides. In this section, we'll explore these operations, which include opening a file, closing a file, changing the way a file is buffered, deleting a file, and renaming a file.

Opening a File

```
FILE *fopen(const char * restrict filename,
            const char * restrict mode);
```

fopen

Opening a file for use as a stream requires a call of the `fopen` function. `fopen`'s first argument is a string containing the name of the file to be opened. (A "file name" may include information about the file's location, such as a drive specifier or path.) The second argument is a "mode string" that specifies what operations we intend to perform on the file. The string `"r"`, for instance, indicates that data will be read from the file, but none will be written to it.

restrict keyword ➤ 17.8

C99

Note that `restrict` appears twice in the prototype for the `fopen` function. `restrict`, which is a C99 keyword, indicates that `filename` and `mode` should point to strings that don't share memory locations. The C89 prototype for `fopen` doesn't contain `restrict` but is otherwise identical. `restrict` has no effect on the behavior of `fopen`, so it can usually just be ignored. In this and subsequent chapters, I'll italicize *`restrict`* as a reminder that it's a C99 feature.

escape sequences ➤ 7.3

Windows programmers: Be careful when the file name in a call of `fopen` includes the \ character, since C treats \ as the beginning of an escape sequence. The call

```
fopen("c:\project\test1.dat", "r")
```

will fail, because the compiler treats `\t` as a character escape. (`\p` isn't a valid character escape, but it looks like one. The C standard states that its meaning is

undefined.) There are two ways to avoid the problem. One is to use \\ instead of \:

```
fopen("c:\\project\\test1.dat", "r")
```

The other technique is even easier—just use the / character instead of \:

```
fopen("c:/project/test1.dat", "r")
```

Windows will happily accept / instead of \ as the directory separator.

`fopen` returns a file pointer that the program can (and usually will) save in a variable and use later whenever it needs to perform an operation on the file. Here's a typical call of `fopen`, where `fp` is a variable of type `FILE *`:

```
fp = fopen("in.dat", "r");   /* opens in.dat for reading */
```

When the program calls an input function to read from `in.dat` later, it will supply `fp` as an argument.

When it can't open a file, `fopen` returns a null pointer. Perhaps the file doesn't exist, or it's in the wrong place, or we don't have permission to open it.

Never assume that a file can be opened; always test the return value of `fopen` to make sure it's not a null pointer.

Modes

Which mode string we'll pass to `fopen` depends not only on what operations we plan to perform on the file later but also on whether the file contains text or binary data. To open a text file, we'd use one of the mode strings in Table 22.2.

Table 22.2 Mode Strings for Text Files

String	*Meaning*
`"r"`	Open for reading
`"w"`	Open for writing (file need not exist)
`"a"`	Open for appending (file need not exist)
`"r+"`	Open for reading and writing, starting at beginning
`"w+"`	Open for reading and writing (truncate if file exists)
`"a+"`	Open for reading and writing (append if file exists)

Q&A

When we use `fopen` to open a binary file, we'll need to include the letter `b` in the mode string. Table 22.3 lists mode strings for binary files.

From Tables 22.2 and 22.3, we see that `<stdio.h>` distinguishes between *writing* data and *appending* data. When data is written to a file, it normally overwrites what was previously there. When a file is opened for appending, however, data written to the file is added at the end, thus preserving the file's original contents.

By the way, special rules apply when a file is opened for both reading and writing (the mode string contains the + character). We can't switch from reading to writ-

Table 22.3
Mode Strings for Binary Files

String	*Meaning*
`"rb"`	Open for reading
`"wb"`	Open for writing (file need not exist)
`"ab"`	Open for appending (file need not exist)
`"r+b"` or `"rb+"`	Open for reading and writing, starting at beginning
`"w+b"` or `"wb+"`	Open for reading and writing (truncate if file exists)
`"a+b"` or `"ab+"`	Open for reading and writing (append if file exists)

file-positioning functions ➤22.7

ing without first calling a file-positioning function unless the reading operation encountered the end of the file. Also, we can't switch from writing to reading without either calling `fflush` (covered later in this section) or calling a file-positioning function.

Closing a File

```
int fclose(FILE *stream);
```

fclose

The `fclose` function allows a program to close a file that it's no longer using. The argument to `fclose` must be a file pointer obtained from a call of `fopen` or `freopen` (discussed later in this section). `fclose` returns zero if the file was closed successfully; otherwise, it returns the error code `EOF` (a macro defined in `<stdio.h>`).

Q&A

To show how `fopen` and `fclose` are used in practice, here's the outline of a program that opens the file `example.dat` for reading, checks that it was opened successfully, then closes it before terminating:

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

#define FILE_NAME "example.dat"

int main(void)
{
  FILE *fp;

  fp = fopen(FILE_NAME, "r");
  if (fp == NULL) {
    printf("Can't open %s\n", FILE_NAME);
    exit(EXIT_FAILURE);
  }
  ...
  fclose(fp);
  return 0;
}
```

Of course, C programmers being the way they are, it's not unusual to see the call of `fopen` combined with the declaration of `fp`:

```
FILE *fp = fopen(FILE_NAME, "r");
```

or the test against NULL:

```
if ((fp = fopen(FILE_NAME, "r")) == NULL) ...
```

Attaching a File to an Open Stream

```
FILE *freopen(const char * restrict filename,
              const char * restrict mode,
              FILE * restrict stream);
```

freopen

`freopen` attaches a different file to a stream that's already open. The most common use of `freopen` is to associate a file with one of the standard streams (`stdin`, `stdout`, or `stderr`). To cause a program to begin writing to the file `foo`, for instance, we could use the following call of `freopen`:

```
if (freopen("foo", "w", stdout) == NULL) {
  /* error; foo can't be opened */
}
```

After closing any file previously associated with `stdout` (by command-line redirection or a previous call of `freopen`), `freopen` will open `foo` and associate it with `stdout`.

`freopen`'s normal return value is its third argument (a file pointer). If it can't open the new file, `freopen` returns a null pointer. (`freopen` ignores the error if the old file can't be closed.)

C99

C99 adds a new twist. If `filename` is a null pointer, `freopen` attempts to change the stream's mode to that specified by the `mode` parameter. Implementations aren't required to support this feature, however: if they do, they may place restrictions on which mode changes are permitted.

Obtaining File Names from the Command Line

When we're writing a program that will need to open a file, one problem soon becomes apparent: how do we supply the file name to the program? Building file names into the program itself doesn't provide much flexibility, and prompting the

Q&A

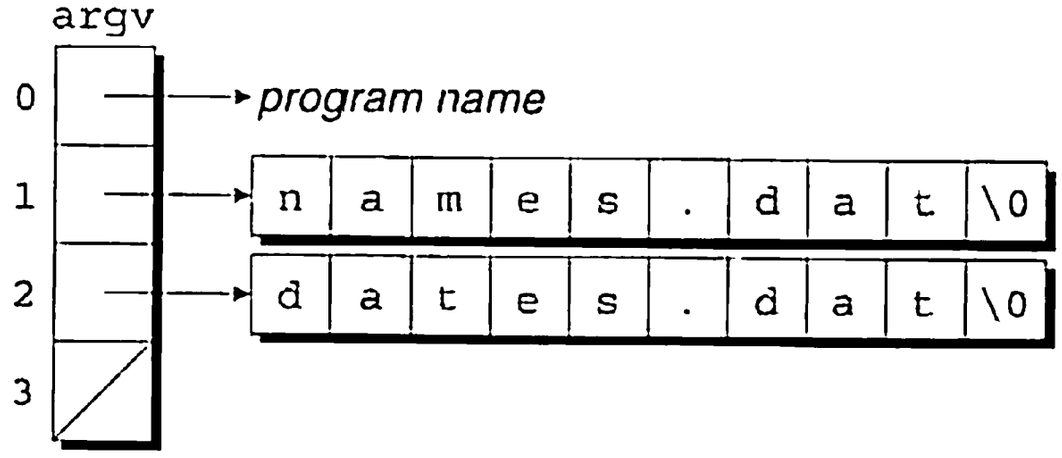
user to enter file names can be awkward. Often, the best solution is to have the program obtain file names from the command line. When we execute a program named `demo`, for example, we might supply it with file names by putting them on the command line:

```
demo names.dat dates.dat
```

In Section 13.7, we saw how to access command-line arguments by defining `main` as a function with two parameters:

```
int main(int argc, char *argv[])
{
  ...
}
```

`argc` is the number of command-line arguments; `argv` is an array of pointers to the argument strings. `argv[0]` points to the program name, `argv[1]` through `argv[argc-1]` point to the remaining arguments, and `argv[argc]` is a null pointer. In the example above, `argc` is 3, `argv[0]` points to a string containing the program name, `argv[1]` points to the string `"names.dat"`, and `argv[2]` points to the string `"dates.dat"`:



PROGRAM

Checking Whether a File Can Be Opened

The following program determines if a file exists and can be opened for reading. When the program is run, the user will give it a file name to check:

canopen *file*

The program will then print either *file* `can be opened` or *file* `can't be opened`. If the user enters the wrong number of arguments on the command line, the program will print the message `usage: canopen filename` to remind the user that `canopen` requires a single file name.

canopen.c

```
/* Checks whether a file can be opened for reading */

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

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

  if (argc != 2) {
    printf("usage: canopen filename\n");
    exit(EXIT_FAILURE);
  }

  if ((fp = fopen(argv[1], "r")) == NULL) {
    printf("%s can't be opened\n", argv[1]);
    exit(EXIT_FAILURE);
  }

  printf("%s can be opened\n", argv[1]);
  fclose(fp);
  return 0;
}
```

Note that we can use redirection to discard the output of `canopen` and simply test the status value it returns.

Temporary Files

```
FILE *tmpfile(void);
char *tmpnam(char *s);
```

Real-world programs often need to create temporary files—files that exist only as long as the program is running. C compilers, for instance, often create temporary files. A compiler might first translate a C program to some intermediate form, which it stores in a file. The compiler would then read the file later as it translates the program to object code. Once the program is completely compiled, there's no need to preserve the file containing the program's intermediate form. `<stdio.h>` provides two functions, `tmpfile` and `tmpnam`, for working with temporary files.

tmpfile

`tmpfile` creates a temporary file (opened in `"wb+"` mode) that will exist until it's closed or the program ends. A call of `tmpfile` returns a file pointer that can be used to access the file later:

```
FILE *tempptr;
...
tempptr = tmpfile();   /* creates a temporary file */
```

If it fails to create a file, `tmpfile` returns a null pointer.

Although `tmpfile` is easy to use, it has a couple of drawbacks: (1) we don't know the name of the file that `tmpfile` creates, and (2) we can't decide later to make the file permanent. If these restrictions turn out to be a problem, the alternative is to create a temporary file using `fopen`. Of course, we don't want this file to have the same name as a previously existing file, so we need some way to generate new file names; that's where the `tmpnam` function comes in.

tmpnam

`tmpnam` generates a name for a temporary file. If its argument is a null pointer, `tmpnam` stores the file name in a static variable and returns a pointer to it:

```
char *filename;
...
filename = tmpnam(NULL);   /* creates a temporary file name */
```

Otherwise, `tmpnam` copies the file name into a character array provided by the programmer:

```
char filename[L_tmpnam];
...
tmpnam(filename);          /* creates a temporary file name */
```

In the latter case, `tmpnam` also returns a pointer to the first character of this array. `L_tmpnam` is a macro in `<stdio.h>` that specifies how long to make a character array that will hold a temporary file name.

Be sure that `tmpnam`'s argument points to an array of at least `L_tmpnam` characters. Also, be careful not to call `tmpnam` too often; the `TMP_MAX` macro (defined in `<stdio.h>`) specifies the maximum number of temporary file names that can potentially be generated by `tmpnam` during the execution of a program. If it fails to generate a file name, `tmpnam` returns a null pointer.

File Buffering

```
int fflush(FILE *stream);
void setbuf(FILE * restrict stream,
            char * restrict buf);
int setvbuf(FILE * restrict stream,
            char * restrict buf,
            int mode, size_t size);
```

Transferring data to or from a disk drive is a relatively slow operation. As a result, it isn't feasible for a program to access a disk file directly each time it wants to read or write a byte. The secret to achieving acceptable performance is *buffering:* data written to a stream is actually stored in a buffer area in memory; when it's full (or the stream is closed), the buffer is "flushed" (written to the actual output device). Input streams can be buffered in a similar way: the buffer contains data from the input device; input is read from this buffer instead of the device itself. Buffering can result in enormous gains in efficiency, since reading a byte from a buffer or storing a byte in a buffer takes hardly any time at all. Of course, it takes time to transfer the buffer contents to or from disk, but one large "block move" is much faster than many tiny byte moves.

The functions in `<stdio.h>` perform buffering automatically when it seems advantageous. The buffering takes place behind the scenes, and we usually don't worry about it. On rare occasions, though, we may need to take a more active role. If so, we can use the functions `fflush`, `setbuf`, and `setvbuf`.

fflush

When a program writes output to a file, the data normally goes into a buffer first. The buffer is flushed automatically when it's full or the file is closed. By calling `fflush`, however, a program can flush a file's buffer as often as it wishes. The call

Q&A

```
fflush(fp);      /* flushes buffer for fp */
```

flushes the buffer for the file associated with `fp`. The call

```
fflush(NULL);    /* flushes all buffers */
```

flushes *all* output streams. `fflush` returns zero if it's successful and `EOF` if an error occurs.

setvbuf

`setvbuf` allows us to change the way a stream is buffered and to control the size and location of the buffer. The function's third argument, which specifies the kind of buffering desired, should be one of the following macros:

- `_IOFBF` (full buffering). Data is read from the stream when the buffer is empty or written to the stream when it's full.
- `_IOLBF` (line buffering). Data is read from the stream or written to the stream one line at a time.
- `_IONBF` (no buffering). Data is read from the stream or written to the stream directly, without a buffer.

(All three macros are defined in `<stdio.h>`.) Full buffering is the default for streams that aren't connected to interactive devices.

`setvbuf`'s second argument (if it's not a null pointer) is the address of the desired buffer. The buffer might have static storage duration, automatic storage duration, or even be allocated dynamically. Making the buffer automatic allows its space to be reclaimed automatically at block exit; allocating it dynamically enables us to free the buffer when it's no longer needed. `setvbuf`'s last argument is the number of bytes in the buffer. A larger buffer may give better performance; a smaller buffer saves space.

For example, the following call of `setvbuf` changes the buffering of `stream` to full buffering, using the `N` bytes in the `buffer` array as the buffer:

```
char buffer[N];
...
setvbuf(stream, buffer, _IOFBF, N);
```

`setvbuf` must be called after `stream` is opened but before any other operations are performed on it.

It's also legal to call `setvbuf` with a null pointer as the second argument, which requests that `setvbuf` create a buffer with the specified size. `setvbuf` returns zero if it's successful. It returns a nonzero value if the mode argument is invalid or the request can't be honored.

setbuf

`setbuf` is an older function that assumes default values for the buffering mode and buffer size. If `buf` is a null pointer, the call `setbuf(stream, buf)` is equivalent to

```
(void) setvbuf(stream, NULL, _IONBF, 0);
```

Otherwise, it's equivalent to

```
(void) setvbuf(stream, buf, _IOFBF, BUFSIZ);
```

where `BUFSIZ` is a macro defined in `<stdio.h>`. The `setbuf` function is considered obsolete; it's not recommended for use in new programs.

When using `setvbuf` or `setbuf`, be sure to close the stream before its buffer is deallocated. In particular, if the buffer is local to a function and has automatic storage duration, be sure to close the stream before the function returns.

Miscellaneous File Operations

```
int remove(const char *filename);
int rename(const char *old, const char *new);
```

The functions `remove` and `rename` allow a program to perform basic file management operations. Unlike most other functions in this section, `remove` and `rename` work with file *names* instead of file *pointers*. Both functions return zero if they succeed and a nonzero value if they fail.

remove

`remove` deletes a file:

```
remove("foo");              /* deletes the file named "foo" */
```

If a program uses `fopen` (instead of `tmpfile`) to create a temporary file, it can use `remove` to delete the file before the program terminates. Be sure that the file to be removed has been closed; the effect of removing a file that's currently open is implementation-defined.

rename

`rename` changes the name of a file:

```
rename("foo", "bar");    /* renames "foo" to "bar" */
```

`rename` is handy for renaming a temporary file created using `fopen` if a program should decide to make it permanent. If a file with the new name already exists, the effect is implementation-defined.

If the file to be renamed is open, be sure to close it before calling `rename`; the function may fail if asked to rename an open file.

22.3 Formatted I/O

In this section, we'll examine library functions that use format strings to control reading and writing. These functions, which include our old friends `printf` and `scanf`, have the ability to convert data from character form to numeric form during input and from numeric form to character form during output. None of the other I/O functions can do such conversions.

The ...printf Functions

```
int fprintf(FILE * restrict stream,
            const char * restrict format, ...);
int printf(const char * restrict format, ...);
```

fprintf
printf

The `fprintf` and `printf` functions write a variable number of data items to an output stream, using a format string to control the appearance of the output.

ellipsis ➤26.1

The prototypes for both functions end with the `...` symbol (an ***ellipsis***), which indicates a variable number of additional arguments. Both functions return the number of characters written: a negative return value indicates that an error occurred.

The only difference between `printf` and `fprintf` is that `printf` always writes to `stdout` (the standard output stream), whereas `fprintf` writes to the stream indicated by its first argument:

```
printf("Total: %d\n", total);          /* writes to stdout */
fprintf(fp, "Total: %d\n", total);     /* writes to fp */
```

A call of `printf` is equivalent to a call of `fprintf` with `stdout` as the first argument.

Don't think of `fprintf` as merely a function that writes data to disk files, though. Like many functions in `<stdio.h>`, `fprintf` works fine with any output stream. In fact, one of the most common uses of `fprintf`—writing error messages to `stderr`, the standard error stream—has nothing to do with disk files. Here's what such a call might look like:

```
fprintf(stderr, "Error: data file can't be opened.\n");
```

Writing the message to `stderr` guarantees that it will appear on the screen even if the user redirects `stdout`.

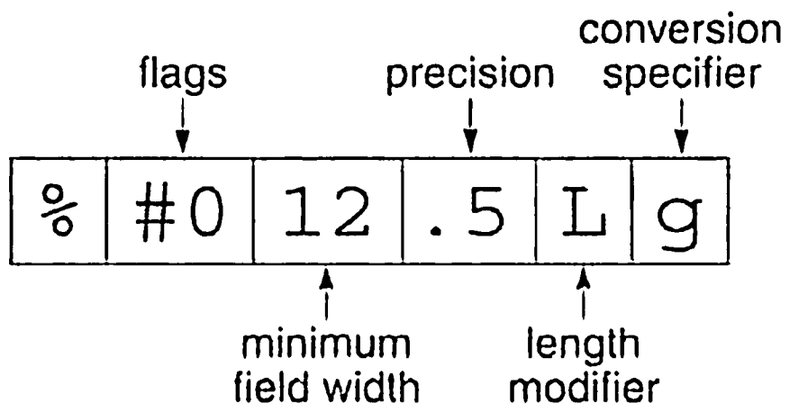
v...printf functions ➤26.1

There are two other functions in `<stdio.h>` that can write formatted output to a stream. These functions, named `vfprintf` and `vprintf`, are fairly obscure. Both rely on the `va_list` type, which is declared in `<stdarg.h>`, so they're discussed along with that header.

...printf Conversion Specifications

Both `printf` and `fprintf` require a format string containing ordinary characters and/or conversion specifications. Ordinary characters are printed as is; conversion specifications describe how the remaining arguments are to be converted to character form for display. Section 3.1 described conversion specifications briefly. and we added more details in later chapters. We'll now review what we know about conversion specifications and fill in the remaining gaps.

A `...printf` conversion specification consists of the `%` character, followed by as many as five distinct items:



Here's a detailed description of these items, which must appear in the order shown:

- ***Flags*** (optional; more than one permitted). The - flag causes left justification within a field; the other flags affect the way numbers are displayed. Table 22.4 gives a complete list of flags.

Table 22.4
Flags for ...printf Functions

Flag	*Meaning*
-	Left-justify within field. (The default is right justification.)
+	Numbers produced by signed conversions always begin with + or -. (Normally, only negative numbers are preceded by a sign.)
space	Nonnegative numbers produced by signed conversions are preceded by a space. (The + flag overrides the *space* flag.)
#	Octal numbers begin with 0, nonzero hexadecimal numbers with 0x or 0X. Floating-point numbers always have a decimal point. Trailing zeros aren't removed from numbers printed with the g or G conversions.
0 *(zero)*	Numbers are padded with leading zeros up to the field width. The 0 flag is ignored if the conversion is d, i, o, u, x, or X and a precision is specified. (The - flag overrides the 0 flag.)

- ***Minimum field width*** (optional). An item that's too small to occupy this number of characters will be padded. (By default, spaces are added to the left of the item, thus right-justifying it within the field.) An item that's too large for the field width will still be displayed in its entirety. The field width is either an integer or the character *. If * is present, the field width is obtained from the next argument. If this argument is negative, it's treated as a positive number preceded by a - flag.
- ***Precision*** (optional). The meaning of the precision depends on the conversion:

  d, i, o, u, x, X: minimum number of digits
  (leading zeros are added if the number has fewer digits)
  a, A, e, E, f, F: number of digits after the decimal point
  g, G: number of significant digits
  s: maximum number of bytes

  The precision is a period (.) followed by an integer or the character *. If * is present, the precision is obtained from the next argument. (If this argument is negative, the effect is the same as not specifying a precision.) If only the period is present, the precision is zero.

- *Length modifier* (optional). The presence of a length modifier indicates that the item to be displayed has a type that's longer or shorter than is normal for a particular conversion specification. (For example, `%d` normally refers to an `int` value; `%hd` is used to display a `short int` and `%ld` is used to display a `long int`.) Table 22.5 lists each length modifier, the conversion specifiers with which it may be used, and the type indicated by the combination of the two. (Any combination of length modifier and conversion specifier not shown in the table causes undefined behavior.)

Table 22.5 Length Modifiers for ...`printf` Functions

Length Modifier	*Conversion Specifiers*	*Meaning*
hh†	d, i, o, u, x, X	signed char, unsigned char
	n	signed char *
h	d, i, o, u, x, X	short int, unsigned short int
	n	short int *
l *(ell)*	d, i, o, u, x, X	long int, unsigned long int
	n	long int *
	c	wint_t
	s	wchar_t *
	a, A, e, E, f, F, g, G	no effect
ll† *(ell-ell)*	d, i, o, u, x, X	long long int, unsigned long long int
	n	long long int *
j†	d, i, o, u, x, X	intmax_t, uintmax_t
	n	intmax_t *
z†	d, i, o, u, x, X	size_t
	n	size_t *
t†	d, i, o, u, x, X	ptrdiff_t
	n	ptrdiff_t *
L	a, A, e, E, f, F, g, G	long double

†C99 only

- *Conversion specifier*. The conversion specifier must be one of the characters listed in Table 22.6. Notice that `f`, `F`, `e`, `E`, `g`, `G`, `a`, and `A` are all designed to write `double` values. However, they work fine with `float` values as well; thanks to the default argument promotions, `float` arguments are converted automatically to `double` when passed to a function with a variable number of arguments. Similarly, a character passed to ...`printf` is converted automatically to `int`, so the `c` conversion works properly.

default argument promotions ➤9.3

Be careful to follow the rules described here; the effect of using an invalid conversion specification is undefined.

Table 22.6 Conversion Specifiers for ...printf Functions

Conversion Specifier	*Meaning*
d, i	Converts an int value to decimal form.
o, u, x, X	Converts an unsigned int value to base 8 (o), base 10 (u), or base 16 (x, X). x displays the hexadecimal digits a–f in lower case; X displays them in upper case.
f, F†	Converts a double value to decimal form, putting the decimal point in the correct position. If no precision is specified, displays six digits after the decimal point.
e, E	Converts a double value to scientific notation. If no precision is specified, displays six digits after the decimal point. If e is chosen, the exponent is preceded by the letter e; if E is chosen, the exponent is preceded by E.
g, G	g converts a double value to either f form or e form. e form is selected if the number's exponent is less than –4 *or* greater than or equal to the precision. Trailing zeros are not displayed (unless the # flag is used); a decimal point appears only when followed by a digit. G chooses between F and E forms.
a†, A†	Converts a double value to hexadecimal scientific notation using the form [-]0x*h*.*hhhh*p±*d*, where [-] is an optional minus sign, the *h*'s represent hex digits, ± is either a plus or minus sign, and *d* is the exponent. *d* is a decimal number that represents a power of 2. If no precision is specified, enough digits are displayed after the decimal point to represent the exact value of the number (if possible). a displays the hex digits a–f in lower case; A displays them in upper case. The choice of a or A also affects the case of the letters x and p.
c	Displays an int value as an unsigned character.
s	Writes the characters pointed to by the argument. Stops writing when the number of bytes specified by the precision (if present) is reached or a null character is encountered.
p	Converts a void * value to printable form.
n	The corresponding argument must point to an object of type int. Stores in this object the number of characters written so far by this call of ...printf; produces no output.
%	Writes the character %.

†C99 only

C99 C99 Changes to ...printf Conversion Specifications

The conversion specifications for printf and fprintf have undergone a number of changes in C99:

- ***Additional length modifiers.*** C99 adds the hh, ll, j, z, and t length modifiers. hh and ll provide additional length options, j allows greatest-width integers to be written, and z and t make it easier to write values of type size_t and ptrdiff_t, respectively.

greatest-width integers ➤27.1

- *Additional conversion specifiers.* C99 adds the `F`, `a`, and `A` conversion specifiers. `F` is the same as `f` except for the way in which infinity and NaN (see below) are written. The `a` and `A` conversion specifications are rarely used. They're related to hexadecimal floating constants, which are discussed in the Q&A section at the end of Chapter 7.

IEEE floating-point standard ➤23.4

- *Ability to write infinity and NaN.* The IEEE 754 floating-point standard allows the result of a floating-point operation to be infinity, negative infinity, or NaN ("not a number"). For example, dividing 1.0 by 0.0 yields positive infinity, dividing –1.0 by 0.0 yields negative infinity, and dividing 0.0 by 0.0 yields NaN (because the result is mathematically undefined). In C99, the `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers are capable of converting these special values to a form that can be displayed. `a`, `e`, `f`, and `g` convert positive infinity to `inf` or `infinity` (either one is legal), negative infinity to `-inf` or `-infinity`, and NaN to `nan` or `-nan` (possibly followed by a series of characters enclosed in parentheses). `A`, `E`, `F`, and `G` are equivalent to `a`, `e`, `f`, and `g`, except that upper-case letters are used (`INF`, `INFINITY`, `NAN`).

wide characters ➤25.2

- *Support for wide characters.* Another C99 feature is the ability of `fprintf` to write wide characters. The `%lc` conversion specification is used to write a single wide character; `%ls` is used for a string of wide characters.
- *Previously undefined conversion specifications now allowed.* In C89, the effect of using `%le`, `%lE`, `%lf`, `%lg`, and `%lG` is undefined. These conversion specifications are legal in C99 (the `l` length modifier is simply ignored).

Examples of ...`printf` Conversion Specifications

Whew! It's about time for a few examples. We've seen plenty of everyday conversion specifications in previous chapters, so we'll concentrate here on illustrating some of the more advanced ones. As in previous chapters, I'll use • to represent the space character.

Let's start off by examining the effect of flags on the `%d` conversion (they have a similar effect on other conversions). The first line of Table 22.7 shows the effect of `%8d` without any flags. The next four lines show the effect of the `-`, `+`, *space*, and `0` flags (the # flag is never used with `%d`). The remaining lines show the effect of combinations of flags.

Table 22.7 Effect of Flags on the `%d` Conversion

Conversion Specification	*Result of Applying Conversion to 123*	*Result of Applying Conversion to –123*
`%8d`	•••••123	••••-123
`%-8d`	123•••••	-123••••
`%+8d`	••••+123	••••-123
`% 8d`	•••••123	••••-123
`%08d`	00000123	-0000123
`%-+8d`	+123••••	-123••••
`%- 8d`	•123••••	-123••••
`%+08d`	+0000123	-0000123
`% 08d`	•0000123	-0000123

Table 22.8 shows the effect of the # flag on the o, x, X, g, and G conversions.

Table 22.8
Effect of the # Flag

Conversion Specification	*Result of Applying Conversion to 123*	*Result of Applying Conversion to 123.0*
%8o	•••••173	
%#8o	••••0173	
%8x	••••••7b	
%#8x	••••0x7b	
%8X	••••••7B	
%#8X	••••0X7B	
%8g		•••••123
%#8g		•123.000
%8G		•••••123
%#8G		•123.000

In previous chapters, we've used the minimum field width and precision when displaying numbers, so there's no point in more examples here. Instead, Table 22.9 shows the effect of the minimum field width and precision on the %s conversion.

Table 22.9
Effect of Minimum Field Width and Precision on the %s Conversion

Conversion Specification	*Result of Applying Conversion to* "bogus"	*Result of Applying Conversion to* "buzzword"
%6s	•bogus	buzzword
%-6s	bogus•	buzzword
%.4s	bogu	buzz
%6.4s	••bogu	••buzz
%-6.4s	bogu••	buzz••

Table 22.10 illustrates how the %g conversion displays some numbers in %e form and others in %f form. All numbers in the table were written using the %.4g conversion specification. The first two numbers have exponents of at least 4, so they're displayed in %e form. The next eight numbers are displayed in %f form. The last two numbers have exponents less than –4, so they're displayed in %e form.

Table 22.10
Examples of the %g Conversion

Number	*Result of Applying %.4g Conversion to Number*
123456.	1.235e+05
12345.6	1.235e+04
1234.56	1235
123.456	123.5
12.3456	12.35
1.23456	1.235
.123456	0.1235
.0123456	0.01235
.00123456	0.001235
.000123456	0.0001235
.0000123456	1.235e-05
.00000123456	1.235e-06

In the past, we've assumed that the minimum field width and precision were constants embedded in the format string. Putting the * character where either number would normally go allows us to specify it as an argument *after* the format string. For example, the following calls of `printf` all produce the same output:

```
printf("%6.4d", i);
printf("%*.4d", 6, i);
printf("%6.*d", 4, i);
printf("%*.*d", 6, 4, i);
```

Notice that the values to be filled in for the * come just before the value to be displayed. A major advantage of *, by the way, is that it allows us to use a macro to specify the width or precision:

```
printf("%*d", WIDTH, i);
```

We can even compute the width or precision during program execution:

```
printf("%*d", page_width / num_cols, i);
```

The most unusual specifications are `%p` and `%n`. The `%p` conversion allows us to print the value of a pointer:

```
printf("%p", (void *) ptr);   /* displays value of ptr */
```

Although `%p` is occasionally useful during debugging, it's not a feature that most programmers use on a daily basis. The C standard doesn't specify what a pointer looks like when printed using `%p`, but it's likely to be shown as an octal or hexadecimal number.

The `%n` conversion is used to find out how many characters have been printed so far by a call of ...`printf`. For example, after the call

```
printf("%d%n\n", 123, &len);
```

the value of `len` will be 3, since `printf` had written 3 characters (`123`) by the time it reached `%n`. Notice that `&` must precede `len` (because `%n` requires a pointer) and that `len` itself isn't printed.

The ...scanf Functions

```
int fscanf(FILE * restrict stream,
           const char * restrict format, ...);
int scanf(const char * restrict format, ...);
```

fscanf
scanf

`fscanf` and `scanf` read data items from an input stream, using a format string to indicate the layout of the input. After the format string, any number of pointers—each pointing to an object—follow as additional arguments. Input items are converted (according to conversion specifications in the format string) and stored in these objects.

`scanf` always reads from `stdin` (the standard input stream), whereas `fscanf` reads from the stream indicated by its first argument:

```
scanf("%d%d", &i, &j);          /* reads from stdin */
fscanf(fp, "%d%d", &i, &j);     /* reads from fp */
```

A call of `scanf` is equivalent to a call of `fscanf` with `stdin` as the first argument.

The ...`scanf` functions return prematurely if an ***input failure*** occurs (no more input characters could be read) or if a ***matching failure*** occurs (the input characters didn't match the format string). (In C99, an input failure can also occur because of an ***encoding error,*** which means that an attempt was made to read a multibyte character, but the input characters didn't correspond to any valid multibyte character.) Both functions return the number of data items that were read and assigned to objects; they return `EOF` if an input failure occurs before any data items can be read.

C99

multibyte characters ➤ 25.2

Loops that test `scanf`'s return value are common in C programs. The following loop, for example, reads a series of integers one by one, stopping at the first sign of trouble:

idiom

```
while (scanf("%d", &i) == 1) {
  ...
}
```

...`scanf` Format Strings

Calls of the ...`scanf` functions resemble those of the ...`printf` functions. That similarity can be misleading, however; the ...`scanf` functions work quite differently from the ...`printf` functions. It pays to think of `scanf` and `fscanf` as "pattern-matching" functions. The format string represents a pattern that a ...`scanf` function attempts to match as it reads input. If the input doesn't match the format string, the function returns as soon as it detects the mismatch; the input character that didn't match is "pushed back" to be read in the future.

A ...`scanf` format string may contain three things:

- ***Conversion specifications.*** Conversion specifications in a ...`scanf` format string resemble those in a ...`printf` format string. Most conversion specifications skip white-space characters at the beginning of an input item (the exceptions are `%[`, `%c`, and `%n`). Conversion specifications never skip *trailing* white-space characters, however. If the input contains `•123¤`, the `%d` conversion specification consumes `•`, `1`, `2`, and `3`, but leaves `¤` unread. (I'm using `•` to represent the space character and `¤` to represent the new-line character.)

white-space characters ➤ 3.2

- ***White-space characters.*** One or more consecutive white-space characters in a ...`scanf` format string match zero or more white-space characters in the input stream.
- ***Non-white-space characters.*** A non-white-space character other than `%` matches the same character in the input stream.

For example, the format string `"ISBN %d-%d-%ld-%d"` specifies that the input will consist of:

the letters `ISBN`
possibly some white-space characters
an integer
the - character
an integer (possibly preceded by white-space characters)
the - character
a long integer (possibly preceded by white-space characters)
the - character
an integer (possibly preceded by white-space characters)

...scanf Conversion Specifications

Conversion specifications for ...`scanf` functions are actually a little simpler than those for ...`printf` functions. A ...`scanf` conversion specification consists of the character `%` followed by the items listed below (in the order shown).

- `*` (optional). The presence of `*` signifies ***assignment suppression:*** an input item is read but not assigned to an object. Items matched using `*` aren't included in the count that ...`scanf` returns.
- ***Maximum field width*** (optional). The maximum field width limits the number of characters in an input item; conversion of the item ends if this number is reached. White-space characters skipped at the beginning of a conversion don't count.
- ***Length modifier*** (optional). The presence of a length modifier indicates that the object in which the input item will be stored has a type that's longer or shorter than is normal for a particular conversion specification. Table 22.11 lists each length modifier, the conversion specifiers with which it may be used, and the type indicated by the combination of the two. (Any combination of length modifier and conversion specifier not shown in the table causes undefined behavior.)

Table 22.11 Length Modifiers for ...`scanf` Functions

Length Modifier	*Conversion Specifiers*	*Meaning*
`hh`†	`d, i, o, u, x, X, n`	`signed char *, unsigned char *`
`h`	`d, i, o, u, x, X, n`	`short int *, unsigned short int *`
`l` *(ell)*	`d, i, o, u, x, X, n`	`long int *, unsigned long int *`
	`a, A, e, E, f, F, g, G`	`double *`
	`c, s, or [`	`wchar_t *`
`ll`† *(ell-ell)*	`d, i, o, u, x, X, n`	`long long int *, unsigned long long int *`
`j`†	`d, i, o, u, x, X, n`	`intmax_t *, uintmax_t *`
`z`†	`d, i, o, u, x, X, n`	`size_t *`
`t`†	`d, i, o, u, x, X, n`	`ptrdiff_t *`
`L`	`a, A, e, E, f, F, g, G`	`long double *`

†C99 only

- *Conversion specifier.* The conversion specifier must be one of the characters listed in Table 22.12.

Table 22.12 Conversion Specifiers for ...scanf Functions

Conversion Specifier	*Meaning*
d	Matches a decimal integer; the corresponding argument is assumed to have type int *.
i	Matches an integer; the corresponding argument is assumed to have type int *. The integer is assumed to be in base 10 unless it begins with 0 (indicating octal) or with 0x or 0X (hexadecimal).
o	Matches an octal integer; the corresponding argument is assumed to have type unsigned int *.
u	Matches a decimal integer; the corresponding argument is assumed to have type unsigned int *.
x, X	Matches a hexadecimal integer; the corresponding argument is assumed to have type unsigned int *.
a†, A†, e, E, f, F†, g, G	Matches a floating-point number; the corresponding argument is assumed to have type float *. In C99, the number can be infinity or NaN.
c	Matches *n* characters, where *n* is the maximum field width, or one character if no field width is specified. The corresponding argument is assumed to be a pointer to a character array (or a character object, if no field width is specified). Doesn't add a null character at the end.
s	Matches a sequence of non-white-space characters, then adds a null character at the end. The corresponding argument is assumed to be a pointer to a character array.
[	Matches a nonempty sequence of characters from a scanset, then adds a null character at the end. The corresponding argument is assumed to be a pointer to a character array.
p	Matches a pointer value in the form that ...printf would have written it. The corresponding argument is assumed to be a pointer to a void * object.
n	The corresponding argument must point to an object of type int. Stores in this object the number of characters read so far by this call of ...scanf. No input is consumed and the return value of ...scanf isn't affected.
%	Matches the character %.

†C99 only

Numeric data items can always begin with a sign (+ or -). The o, u, x, and X specifiers convert the item to unsigned form, however, so they're not normally used to read negative numbers.

The [specifier is a more complicated (and more flexible) version of the s specifier. A complete conversion specification using [has the form %[*set*] or %[^*set*], where *set* can be any set of characters. (If] is one of the characters in *set*, however, it must come first.) %[*set*] matches any sequence of characters in *set* (the *scanset*). %[^*set*] matches any sequence of characters *not* in *set* (in other words, the scanset consists of all characters not in *set*). For example, %[abc]

matches any string containing only the letters a, b, and c, while %[^abc] matches any string that doesn't contain a, b, or c.

Many of the ...scanf conversion specifiers are closely related to the numeric conversion functions in <stdlib.h>. These functions convert strings (like "-297") to their equivalent numeric values (–297). The d specifier, for example, looks for an optional + or - sign, followed by a series of decimal digits; this is exactly the same form that the strtol function requires when asked to convert a string to a decimal number. Table 22.13 shows the correspondence between conversion specifiers and numeric conversion functions.

numeric conversion functions ➤ 26.2

Table 22.13 Correspondence between ...scanf Conversion Specifiers and Numeric Conversion Functions

Conversion Specifier	*Numeric Conversion Function*
d	strtol with 10 as the base
i	strtol with 0 as the base
o	strtoul with 8 as the base
u	strtoul with 10 as the base
x, X	strtoul with 16 as the base
a, A, e, E, f, F, g, G	strtod

It pays to be careful when writing calls of scanf. An invalid conversion specification in a scanf format string is just as bad as one in a printf format string; either one causes undefined behavior.

C99 C99 Changes to ...scanf Conversion Specifications

The conversion specifications for scanf and fscanf have undergone some changes in C99, but the list isn't as extensive as it was for the ...printf functions:

- *Additional length modifiers.* C99 adds the hh, ll, j, z, and t length modifiers. These correspond to the length modifiers in ...printf conversion specifications.
- *Additional conversion specifiers.* C99 adds the F, a, and A conversion specifiers. They're provided for symmetry with ...printf; the ...scanf functions treat them the same as e, E, f, g, and G.
- *Ability to read infinity and NaN.* Just as the ...printf functions can write infinity and NaN, the ...scanf functions can read these values. To be read properly, they should have the same appearance as values written by the ...printf functions, with case being ignored. (For example, either INF or inf will be read as infinity.)
- *Support for wide characters.* The ...scanf functions are able to read multibyte characters, which are then converted to wide characters for storage. The %lc conversion specification is used to read a single multibyte character or a

sequence of multibyte characters; `%ls` is used to read a string of multibyte characters (a null character is added at the end). The `%l` [*set*] and `%l` [^*set*] conversion specifications can also read a string of multibyte characters.

`scanf` Examples

The next three tables contain sample calls of `scanf`. Each call is applied to the input characters shown to its right. Characters printed in ~~strikeout~~ are consumed by the call. The values of the variables after the call appear to the right of the input.

The examples in Table 22.14 show the effect of combining conversion specifications, white-space characters, and non-white-space characters. In three cases no value is assigned to `j`, so it retains its value from before the call of `scanf`. The examples in Table 22.15 show the effect of assignment suppression and specifying a field width. The examples in Table 22.16 illustrate the more esoteric conversion specifiers (`i`, `[`, and `n`).

Table 22.14 `scanf` Examples (Group 1)

`scanf` *Call*	*Input*	*Variables*
`n = scanf("%d%d", &i, &j);`	~~12~~•,•34␣	n: 1 i: 12 j: unchanged
`n = scanf("%d,%d", &i, &j);`	~~12~~•,•34␣	n: 1 i: 12 j: unchanged
`n = scanf("%d ,%d", &i, &j);`	~~12•,•34~~␣	n: 2 i: 12 j: 34
`n = scanf("%d, %d", &i, &j);`	~~12~~•,•34␣	n: 1 i: 12 j: unchanged

Table 22.15 `scanf` Examples (Group 2)

`scanf` *Call*	*Input*	*Variables*
`n = scanf("%*d%d", &i);`	~~12•34~~␣	n: 1 i: 34
`n = scanf("%*s%s", str);`	~~My•Fair~~•Lady␣	n: 1 str: "Fair"
`n = scanf("%1d%2d%3d", &i, &j, &k);`	~~12345~~␣	n: 3 i: 1 j: 23 k: 45
`n = scanf("%2d%2s%2d", &i, str, &j);`	~~123456~~␣	n: 3 i: 12 str: "34" j: 56

Table 22.16 scanf Examples (Group 3)

scanf *Call*	*Input*	*Variables*
`n = scanf("%i%i%i", &i, &j, &k);`	~~12•012•0x12~~␣	n: 3 i: 12 j: 10 k: 18
`n = scanf("%[0123456789]", str);`	~~123~~abc␣	n: 1 str: "123"
`n = scanf("%[0123456789]", str);`	abc123␣	n: 0 str: unchanged
`n = scanf("%[^0123456789]", str);`	~~abc~~123␣	n: 1 str: "abc"
`n = scanf("%*d%d%n", &i, &j);`	~~10•20~~•30␣	n: 1 i: 20 j: 5

Detecting End-of-File and Error Conditions

```
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
```

If we ask a ...`scanf` function to read and store *n* data items, we expect its return value to be *n*. If the return value is less than *n*, something went wrong. There are three possibilities:

- ***End-of-file.*** The function encountered end-of-file before matching the format string completely.
- ***Read error.*** The function was unable to read characters from the stream.
- ***Matching failure.*** A data item was in the wrong format. For example, the function might have encountered a letter while searching for the first digit of an integer.

But how can we tell which kind of failure occurred? In many cases, it doesn't matter; something went wrong, and we've got to abandon the program. There may be times, however, when we'll need to pinpoint the reason for the failure.

Every stream has two indicators associated with it: an ***error indicator*** and an ***end-of-file indicator***. These indicators are cleared when the stream is opened. Not surprisingly, encountering end-of-file sets the end-of-file indicator, and a read error sets the error indicator. (The error indicator is also set when a write error occurs on an output stream.) A matching failure doesn't change either indicator.

clearerr

Once the error or end-of-file indicator is set, it remains in that state until it's explicitly cleared, perhaps by a call of the `clearerr` function. `clearerr` clears both the end-of-file and error indicators:

```
clearerr(fp);     /* clears eof and error indicators for fp */
```

`clearerr` isn't needed often, since some of the other library functions clear one or both indicators as a side effect.

Q&A

feof
ferror

We can call the `feof` and `ferror` functions to test a stream's indicators to determine why a prior operation on the stream failed. The call `feof(fp)` returns a nonzero value if the end-of-file indicator is set for the stream associated with `fp`. The call `ferror(fp)` returns a nonzero value if the error indicator is set. Both functions return zero otherwise.

Q&A

When `scanf` returns a smaller-than-expected value, we can use `feof` and `ferror` to determine the reason. If `feof` returns a nonzero value, we've reached the end of the input file. If `ferror` returns a nonzero value, a read error occurred during input. If neither returns a nonzero value, a matching failure must have occurred. Regardless of what the problem was, the return value of `scanf` tells us how many data items were read before the problem occurred.

To see how `feof` and `ferror` might be used, let's write a function that searches a file for a line that begins with an integer. Here's how we intend to call the function:

```
n = find_int("foo");
```

`"foo"` is the name of the file to be searched. The function returns the value of the integer that it finds, which is then assigned to `n`. If a problem arises—the file can't be opened, a read error occurs, or no line begins with an integer—`find_int` will return an error code (–1, –2, or –3, respectively). I'll assume that no line in the file begins with a negative integer.

```
int find_int(const char *filename)
{
  FILE *fp = fopen(filename, "r");
  int n;

  if (fp == NULL)
    return -1;                  /* can't open file */

  while (fscanf(fp, "%d", &n) != 1) {
    if (ferror(fp)) {
      fclose(fp);
      return -2;                /* read error */
    }
    if (feof(fp)) {
      fclose(fp);
      return -3;                /* integer not found */
    }
    fscanf(fp, "%*[^\n]");      /* skips rest of line */
  }

  fclose(fp);
  return n;
}
```

The `while` loop's controlling expression calls `fscanf` in an attempt to read an integer from the file. If the attempt fails (`fscanf` returns a value other than 1),

`find_int` calls `ferror` and `feof` to see if the problem was a read error or end-of-file. If not, `fscanf` must have failed because of a matching error, so `find_int` skips the rest of the characters on the current line and tries again. Note the use of the conversion `%*[^\n]` to skip all characters up to the next new-line. (Now that we know about scansets, it's time to show off!)

22.4 Character I/O

In this section, we'll examine library functions that read and write single characters. These functions work equally well with text streams and binary streams.

You'll notice that the functions in this section treat characters as values of type `int`, not `char`. One reason is that the input functions indicate an end-of-file (or error) condition by returning `EOF`, which is a negative integer constant.

Output Functions

```
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

putchar `putchar` writes one character to the `stdout` stream:

```
putchar(ch);      /* writes ch to stdout */
```

fputc
putc `fputc` and `putc` are more general versions of `putchar` that write a character to an arbitrary stream:

```
fputc(ch, fp);    /* writes ch to fp */
putc(ch, fp);     /* writes ch to fp */
```

Although `putc` and `fputc` do the same thing, `putc` is usually implemented as a macro (as well as a function), while `fputc` is implemented only as a function. `putchar` itself is usually a macro defined in the following way:

```
#define putchar(c) putc((c), stdout)
```

It may seem odd that the library provides both `putc` and `fputc`. But, as we saw in Section 14.3, macros have several potential problems. The C standard allows the `putc` macro to evaluate the `stream` argument more than once, which `fputc` isn't permitted to do. Although programmers usually prefer `putc`, which gives a faster program, `fputc` is available as an alternative.

Q&A

If a write error occurs, all three functions set the error indicator for the stream and return `EOF`; otherwise, they return the character that was written.

Input Functions

```
int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);
```

getchar

`getchar` reads a character from the `stdin` stream:

```
ch = getchar();    /* reads a character from stdin */
```

fgetc
getc

`fgetc` and `getc` read a character from an arbitrary stream:

```
ch = fgetc(fp);    /* reads a character from fp */
ch = getc(fp);     /* reads a character from fp */
```

All three functions treat the character as an `unsigned char` value (which is then converted to `int` type before it's returned). As a result, they never return a negative value other than EOF.

The relationship between `getc` and `fgetc` is similar to that between `putc` and `fputc`. `getc` is usually implemented as a macro (as well as a function), while `fgetc` is implemented only as a function. `getchar` is normally a macro as well:

```
#define getchar() getc(stdin)
```

For reading characters from a file, programmers usually prefer `getc` over `fgetc`. Since `getc` is normally available in macro form, it tends to be faster. `fgetc` can be used as a backup if `getc` isn't appropriate. (The standard allows the `getc` macro to evaluate its argument more than once, which may be a problem.)

The `fgetc`, `getc`, and `getchar` functions behave the same if a problem occurs. At end-of-file, they set the stream's end-of-file indicator and return EOF. If a read error occurs, they set the stream's error indicator and return EOF. To differentiate between the two situations, we can call either `feof` or `ferror`.

One of the most common uses of `fgetc`, `getc`, and `getchar` is to read characters from a file, one by one, until end-of-file occurs. It's customary to use the following `while` loop for that purpose:

idiom

```
while ((ch = getc(fp)) != EOF) {
  ...
}
```

After reading a character from the file associated with `fp` and storing it in the variable `ch` (which must be of type `int`), the `while` test compares `ch` with EOF. If `ch` isn't equal to EOF, we're not at the end of the file yet, so the body of the loop is executed. If `ch` is equal to EOF, the loop terminates.

Always store the return value of `fgetc`, `getc`, or `getchar` in an `int` variable, not a `char` variable. Testing a `char` variable against `EOF` may give the wrong result.

ungetc

There's one other character input function, `ungetc`, which "pushes back" a character read from a stream and clears the stream's end-of-file indicator. This capability can be handy if we need a "lookahead" character during input. For instance, to read a series of digits, stopping at the first nondigit, we could write

isdigit function ➤23.5

```
while (isdigit(ch = getc(fp))) {
  ...
}
ungetc(ch, fp);   /* pushes back last character read */
```

The number of characters that can be pushed back by consecutive calls of `ungetc`—with no intervening read operations—depends on the implementation and the type of stream involved; only the first call is guaranteed to succeed. Calling a file-positioning function (`fseek`, `fsetpos`, or `rewind`) causes the pushed-back characters to be lost.

file-positioning functions ➤22.7

`ungetc` returns the character it was asked to push back. However, it returns `EOF` if an attempt is made to push back `EOF` or to push back more characters than the implementation allows.

PROGRAM

Copying a File

The following program makes a copy of a file. The names of the original file and the new file will be specified on the command line when the program is executed. For example, to copy the file `f1.c` to `f2.c`, we'd use the command

```
fcopy f1.c f2.c
```

`fcopy` will issue an error message if there aren't exactly two file names on the command line or if either file can't be opened.

fcopy.c

```
/* Copies a file */

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

int main(int argc, char *argv[])
{
  FILE *source_fp, *dest_fp;
  int ch;
```

```
if (argc != 3) {
  fprintf(stderr, "usage: fcopy source dest\n");
  exit(EXIT_FAILURE);
}

if ((source_fp = fopen(argv[1], "rb")) == NULL) {
  fprintf(stderr, "Can't open %s\n", argv[1]);
  exit(EXIT_FAILURE);
}

if ((dest_fp = fopen(argv[2], "wb")) == NULL) {
  fprintf(stderr, "Can't open %s\n", argv[2]);
  fclose(source_fp);
  exit(EXIT_FAILURE);
}

while ((ch = getc(source_fp)) != EOF)
  putc(ch, dest_fp);

fclose(source_fp);
fclose(dest_fp);
return 0;
}
```

Using `"rb"` and `"wb"` as the file modes enables `fcopy` to copy both text and binary files. If we used `"r"` and `"w"` instead, the program wouldn't necessarily be able to copy binary files.

22.5 Line I/O

We'll now turn to library functions that read and write lines. These functions are used mostly with text streams, although it's legal to use them with binary streams as well.

Output Functions

```
int fputs(const char * restrict s,
          FILE * restrict stream);
int puts(const char *s);
```

puts

We encountered the `puts` function in Section 13.3; it writes a string of characters to `stdout`:

```
puts("Hi, there!");   /* writes to stdout */
```

After it writes the characters in the string, `puts` always adds a new-line character.

fputs

`fputs` is a more general version of `puts`. Its second argument indicates the stream to which the output should be written:

```
fputs("Hi, there!", fp);   /* writes to fp */
```

Unlike `puts`, the `fputs` function doesn't write a new-line character unless one is present in the string.

Both functions return `EOF` if a write error occurs; otherwise, they return a nonnegative number.

Input Functions

```
char *fgets(char * restrict s, int n,
            FILE * restrict stream);
char *gets(char *s);
```

gets

The `gets` function, which we first encountered in Section 13.3, reads a line of input from `stdin`:

```
gets(str);   /* reads a line from stdin */
```

`gets` reads characters one by one, storing them in the array pointed to by `str`, until it reads a new-line character (which it discards).

fgets

`fgets` is a more general version of `gets` that can read from any stream. `fgets` is also safer than `gets`, since it limits the number of characters that it will store. Here's how we might use `fgets`, assuming that `str` is the name of a character array:

```
fgets(str, sizeof(str), fp);   /* reads a line from fp */
```

This call will cause `fgets` to read characters until it reaches the first new-line character or `sizeof(str) - 1` characters have been read, whichever happens first. If it reads the new-line character, `fgets` stores it along with the other characters. (Thus, `gets` *never* stores the new-line character, but `fgets` *sometimes* does.)

Both `gets` and `fgets` return a null pointer if a read error occurs or they reach the end of the input stream before storing any characters. (As usual, we can call `feof` or `ferror` to determine which situation occurred.) Otherwise, both return their first argument, which points to the array in which the input was stored. As you'd expect, both functions store a null character at the end of the string.

Now that you know about `fgets`, I'd suggest using it instead of `gets` in most situations. With `gets`, there's always the possibility of stepping outside the bounds of the receiving array, so it's safe to use only when the string being read is *guaranteed* to fit into the array. When there's no guarantee (and there usually isn't), it's much safer to use `fgets`. Note that `fgets` will read from the standard input stream if passed `stdin` as its third argument:

```
fgets(str, sizeof(str), stdin);
```

22.6 Block I/O

```
size_t fread(void * restrict ptr,
             size_t size, size_t nmemb,
             FILE * restrict stream);
size_t fwrite(const void * restrict ptr,
              size_t size, size_t nmemb,
              FILE * restrict stream);
```

The `fread` and `fwrite` functions allow a program to read and write large blocks of data in a single step. `fread` and `fwrite` are used primarily with binary streams, although—with care—it's possible to use them with text streams as well.

Q&A

fwrite

`fwrite` is designed to copy an array from memory to a stream. The first argument in a call of `fwrite` is the array's address, the second argument is the size of each array element (in bytes), and the third argument is the number of elements to write. The fourth argument is a file pointer, indicating where the data should be written. To write the entire contents of the array `a`, for instance, we could use the following call of `fwrite`:

```
fwrite(a, sizeof(a[0]), sizeof(a) / sizeof(a[0]), fp);
```

There's no rule that we have to write the entire array; we could just as easily write any portion of it. `fwrite` returns the number of elements (*not* bytes) actually written. This number will be less than the third argument if a write error occurs.

fread

`fread` will read the elements of an array from a stream. `fread`'s arguments are similar to `fwrite`'s: the array's address, the size of each element (in bytes), the number of elements to read, and a file pointer. To read the contents of a file into the array `a`, we might use the following call of `fread`:

```
n = fread(a, sizeof(a[0]), sizeof(a) / sizeof(a[0]), fp);
```

It's important to check `fread`'s return value, which indicates the actual number of elements (*not* bytes) read. This number should equal the third argument unless the end of the input file was reached or a read error occurred. The `feof` and `ferror` functions can be used to determine the reason for any shortage.

Be careful not to confuse `fread`'s second and third arguments. Consider the following call of `fread`:

```
fread(a, 1, 100, fp)
```

We're asking `fread` to read 100 one-byte elements, so it will return a value

between 0 and 100. The following call asks `fread` to read one block of 100 bytes:

```
fread(a, 100, 1, fp)
```

`fread`'s return value in this case will be either 0 or 1.

`fwrite` is convenient for a program that needs to store data in a file before terminating. Later, the program (or another program, for that matter) can use `fread` to read the data back into memory. Despite appearances, the data doesn't need to be in array form; `fread` and `fwrite` work just as well with variables of all kinds. Structures, in particular, can be read by `fread` or written by `fwrite`. To write a structure variable `s` to a file, for instance, we could use the following call of `fwrite`:

```
fwrite(&s, sizeof(s), 1, fp);
```

Be careful when using `fwrite` to write out structures that contain pointer values; these values aren't guaranteed to be valid when read back in.

22.7 File Positioning

```
int fgetpos(FILE * restrict stream,
            fpos_t * restrict pos);
int fseek(FILE *stream, long int offset, int whence);
int fsetpos(FILE *stream, const fpos_t *pos);
long int ftell(FILE *stream);
void rewind(FILE *stream);
```

Every stream has an associated *file position*. When a file is opened, the file position is set at the beginning of the file. (If the file is opened in "append" mode, however, the initial file position may be at the beginning or end of the file, depending on the implementation.) Then, when a read or write operation is performed, the file position advances automatically, allowing us to move through the file in a sequential manner.

Although sequential access is fine for many applications, some programs need the ability to jump around within a file, accessing some data here and other data there. If a file contains a series of records, for example, we might want to jump directly to a particular record and read it or update it. `<stdio.h>` supports this form of access by providing five functions that allow a program to determine the current file position or to change it.

fseek

The `fseek` function changes the file position associated with the first argument (a file pointer). The third argument specifies whether the new position is to

be calculated with respect to the beginning of the file, the current position, or the end of the file. `<stdio.h>` defines three macros for this purpose:

`SEEK_SET`	Beginning of file
`SEEK_CUR`	Current file position
`SEEK_END`	End of file

The second argument is a (possibly negative) byte count. To move to the beginning of a file, for example, the seek direction would be `SEEK_SET` and the byte count would be zero:

```
fseek(fp, 0L, SEEK_SET);      /* moves to beginning of file */
```

To move to the end of a file, the seek direction would be `SEEK_END`:

```
fseek(fp, 0L, SEEK_END);      /* moves to end of file */
```

To move back 10 bytes, the seek direction would be `SEEK_CUR` and the byte count would be –10:

```
fseek(fp, -10L, SEEK_CUR);    /* moves back 10 bytes */
```

Note that the byte count has type `long int`, so I've used `0L` and `-10L` as arguments. (`0` and `-10` would also work, of course, since arguments are converted to the proper type automatically.)

Normally, `fseek` returns zero. If an error occurs (the requested position doesn't exist, for example), `fseek` returns a nonzero value.

The file-positioning functions are best used with binary streams, by the way. C doesn't prohibit programs from using them with text streams, but care is required because of operating system differences. `fseek` in particular is sensitive to whether a stream is text or binary. For text streams, either (1) `offset` (`fseek`'s second argument) must be zero or (2) `whence` (its third argument) must be `SEEK_SET` and `offset` a value obtained by a previous call of `ftell`. (In other words, we can only use `fseek` to move to the beginning or end of a text stream or to return to a place that was visited previously.) For binary streams, `fseek` isn't required to support calls in which `whence` is `SEEK_END`.

ftell

errno variable ➤*24.2*

The `ftell` function returns the current file position as a long integer. (If an error occurs, `ftell` returns `-1L` and stores an error code in `errno`.) The value returned by `ftell` may be saved and later supplied to a call of `fseek`, making it possible to return to a previous file position:

```
long file_pos;
...
file_pos = ftell(fp);              /* saves current position */
...
fseek(fp, file_pos, SEEK_SET);     /* returns to old position */
```

If `fp` is a binary stream, the call `ftell(fp)` returns the current file position as a byte count, where zero represents the beginning of the file. If `fp` is a text stream, however, `ftell(fp)` isn't necessarily a byte count. As a result, it's best not to perform arithmetic on values returned by `ftell`. For example, it's not a good

idea to subtract values returned by `ftell` to see how far apart two file positions are.

rewind

The `rewind` function sets the file position at the beginning. The call `rewind(fp)` is nearly equivalent to `fseek(fp, 0L, SEEK_SET)`. The difference? `rewind` doesn't return a value but does clear the error indicator for `fp`.

fgetpos
fsetpos

Q&A

`fseek` and `ftell` have one problem: they're limited to files whose positions can be stored in a long integer. For working with very large files, C provides two additional functions: `fgetpos` and `fsetpos`. These functions can handle large files because they use values of type `fpos_t` to represent file positions. An `fpos_t` value isn't necessarily an integer: it could be a structure, for instance.

The call `fgetpos(fp, &file_pos)` stores the file position associated with `fp` in the `file_pos` variable. The call `fsetpos(fp, &file_pos)` sets the file position for `fp` to be the value stored in `file_pos`. (This value must have been obtained by a previous call of `fgetpos`.) If a call of `fgetpos` or `fsetpos` fails, it stores an error code in `errno`. Both functions return zero when they succeed and a nonzero value when they fail.

Here's how we might use `fgetpos` and `fsetpos` to save a file position and return to it later:

```
fpos_t file_pos;
...
fgetpos(fp, &file_pos);   /* saves current position */
...
fsetpos(fp, &file_pos);   /* returns to old position */
```

PROGRAM

Modifying a File of Part Records

The following program opens a binary file containing `part` structures, reads the structures into an array, sets the `on_hand` member of each structure to 0, and then writes the structures back to the file. Note that the program opens the file in `"rb+"` mode, allowing both reading and writing.

invclear.c

```
/* Modifies a file of part records by setting the quantity
   on hand to zero for all records */

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

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} inventory[MAX_PARTS];
```

```
int num_parts;

int main(void)
{
  FILE *fp;
  int i;

  if ((fp = fopen("inventory.dat", "rb+")) == NULL) {
    fprintf(stderr, "Can't open inventory file\n");
    exit(EXIT_FAILURE);
  }

  num_parts = fread(inventory, sizeof(struct part),
                    MAX_PARTS, fp);

  for (i = 0; i < num_parts; i++)
    inventory[i].on_hand = 0;

  rewind(fp);
  fwrite(inventory, sizeof(struct part), num_parts, fp);
  fclose(fp);

  return 0;
}
```

Calling `rewind` is critical, by the way. After the `fread` call, the file position is at the end of the file. If we were to call `fwrite` without calling `rewind` first, `fwrite` would add new data to the end of the file instead of overwriting the old data.

22.8 String I/O

The functions described in this section are a bit unusual, since they have nothing to do with streams or files. Instead, they allow us to read and write data using a string as though it were a stream. The `sprintf` and `snprintf` functions write characters into a string in the same way they would be written to a stream; the `sscanf` function reads characters from a string as though it were reading from a stream. These functions, which closely resemble `printf` and `scanf`, are quite useful. `sprintf` and `snprintf` give us access to `printf`'s formatting capabilities without actually having to write data to a stream. Similarly, `sscanf` gives us access to `scanf`'s powerful pattern-matching capabilities. The remainder of this section covers `sprintf`, `snprintf`, and `sscanf` in detail.

Three similar functions (`vsprintf`, `vsnprintf`, and `vsscanf`) also belong to `<stdio.h>`. However, these functions rely on the `va_list` type, which is declared in `<stdarg.h>`. I'll postpone discussing them until Section 26.1, which covers that header.