17 Advanced Uses of Pointers

One can only display complex information in the mind.
Like seeing, movement or flow or alteration of view is more
important than the static picture, no matter how lovely.

In previous chapters, we've seen two important uses of pointers. Chapter 11 showed how using a pointer to a variable as a function argument allows the function to modify the variable. Chapter 12 showed how to process arrays by performing arithmetic on pointers to array elements. This chapter completes our coverage of pointers by examining two additional applications: dynamic storage allocation and pointers to functions.

Using dynamic storage allocation, a program can obtain blocks of memory as needed during execution. Section 17.1 explains the basics of dynamic storage allocation. Section 17.2 discusses dynamically allocated strings, which provide more flexibility than ordinary character arrays. Section 17.3 covers dynamic storage allocation for arrays in general. Section 17.4 deals with the issue of storage deallocation—releasing blocks of dynamically allocated memory when they're no longer needed.

Dynamically allocated structures play a big role in C programming, since they can be linked together to form lists, trees, and other highly flexible data structures. Section 17.5 focuses on linked lists, the most fundamental linked data structure. One of the issues that arises in this section—the concept of a "pointer to a pointer"—is important enough to warrant a section of its own (Section 17.6).

Section 17.7 introduces pointers to functions, a surprisingly useful concept. Some of C's most powerful library functions expect function pointers as arguments. We'll examine one of these functions, `qsort`, which is capable of sorting any array.

The last two sections discuss pointer-related features that first appeared in C99: restricted pointers (Section 17.8) and flexible array members (Section 17.9). These features are primarily of interest to advanced C programmers, so both sections can be safely be skipped by the beginner.

17.1 Dynamic Storage Allocation

variable-length arrays ➤8.3

C's data structures are normally fixed in size. For example, the number of elements in an array is fixed once the program has been compiled. (In C99, the length of a variable-length array is determined at run time, but it remains fixed for the rest of the array's lifetime.) Fixed-size data structures can be a problem, since we're forced to choose their sizes when writing a program; we can't change the sizes without modifying the program and compiling it again.

Consider the `inventory` program of Section 16.3, which allows the user to add parts to a database. The database is stored in an array of length 100. To enlarge the capacity of the database, we can increase the size of the array and recompile the program. But no matter how large we make the array, there's always the possibility that it will fill up. Fortunately, all is not lost. C supports ***dynamic storage allocation:*** the ability to allocate storage during program execution. Using dynamic storage allocation, we can design data structures that grow (and shrink) as needed.

Although it's available for all types of data, dynamic storage allocation is used most often for strings, arrays, and structures. Dynamically allocated structures are of particular interest, since we can link them together to form lists, trees, and other data structures.

Memory Allocation Functions

<stdlib.h> header ➤26.2

To allocate storage dynamically, we'll need to call one of the three memory allocation functions declared in the `<stdlib.h>` header:

- `malloc`—Allocates a block of memory but doesn't initialize it.
- `calloc`—Allocates a block of memory and clears it.
- `realloc`—Resizes a previously allocated block of memory.

Of the three, `malloc` is the most used. It's more efficient than `calloc`, since it doesn't have to clear the memory block that it allocates.

When we call a memory allocation function to request a block of memory, the function has no idea what type of data we're planning to store in the block, so it can't return a pointer to an ordinary type such as `int` or `char`. Instead, the function returns a value of type `void *`. A `void *` value is a "generic" pointer—essentially, just a memory address.

Null Pointers

When a memory allocation function is called, there's always a possibility that it won't be able to locate a block of memory large enough to satisfy our request. If

that should happen, the function will return a ***null pointer.*** A null pointer is a "pointer to nothing"—a special value that can be distinguished from all valid pointers. After we've stored the function's return value in a pointer variable, we must test to see if it's a null pointer.

It's the programmer's responsibility to test the return value of any memory allocation function and take appropriate action if it's a null pointer. The effect of attempting to access memory through a null pointer is undefined; the program may crash or behave unpredictably.

The null pointer is represented by a macro named `NULL`, so we can test `malloc`'s return value in the following way:

```
p = malloc(10000);
if (p == NULL) {
  /* allocation failed; take appropriate action */
}
```

Some programmers combine the call of `malloc` with the `NULL` test:

```
if ((p = malloc(10000)) == NULL) {
  /* allocation failed; take appropriate action */
}
```

The `NULL` macro is defined in six headers: `<locale.h>`, `<stddef.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, and `<time.h>`. (The C99 header `<wchar.h>` also defines `NULL`.) As long as one of these headers is included in a program, the compiler will recognize `NULL`. A program that uses any of the memory allocation functions will include `<stdlib.h>`, of course, making `NULL` available.

In C, pointers test true or false in the same way as numbers. All non-null pointers test true; only null pointers are false. Thus, instead of writing

```
if (p == NULL) …
```

we could write

```
if (!p) …
```

and instead of writing

```
if (p != NULL) …
```

we could write

```
if (p) …
```

As a matter of style, I prefer the explicit comparison with `NULL`.

17.2 Dynamically Allocated Strings

Dynamic storage allocation is often useful for working with strings. Strings are stored in character arrays, and it can be hard to anticipate how long these arrays need to be. By allocating strings dynamically, we can postpone the decision until the program is running.

Using `malloc` to Allocate Memory for a String

The `malloc` function has the following prototype:

```
void *malloc(size_t size);
```

`malloc` allocates a block of `size` bytes and returns a pointer to it. Note that `size` has type `size_t`, an unsigned integer type defined in the C library. Unless we're allocating a very large block of memory, we can just think of `size` as an ordinary integer.

size_t type ➤7.6

Using `malloc` to allocate memory for a string is easy, because C guarantees that a `char` value requires exactly one byte of storage (`sizeof(char)` is 1, in other words). To allocate space for a string of `n` characters, we'd write

```
p = malloc(n + 1);
```

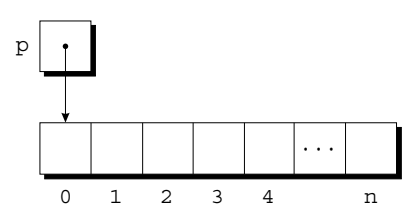
where `p` is a `char *` variable. (The argument is `n + 1` rather than `n` to allow room for the null character.) The generic pointer that `malloc` returns will be converted to `char *` when the assignment is performed; no cast is necessary. (In general, we can assign a `void *` value to a variable of any pointer type and vice versa.) Nevertheless, some programmers prefer to cast `malloc`'s return value:

Q&A

```
p = (char *) malloc(n + 1);
```

When using `malloc` to allocate space for a string, don't forget to include room for the null character.

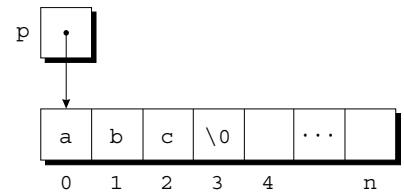
Memory allocated using `malloc` isn't cleared or initialized in any way, so `p` will point to an uninitialized array of `n` + 1 characters:



Calling `strcpy` is one way to initialize this array:

```
strcpy(p, "abc");
```

The first four characters in the array will now be `a`, `b`, `c`, and `\0`:



Using Dynamic Storage Allocation in String Functions

Dynamic storage allocation makes it possible to write functions that return a pointer to a "new" string—a string that didn't exist before the function was called. Consider the problem of writing a function that concatenates two strings without changing either one. C's standard library doesn't include such a function (`strcat` isn't quite what we want, since it modifies one of the strings passed to it), but we can easily write our own.

Our function will measure the lengths of the two strings to be concatenated, then call `malloc` to allocate just the right amount of space for the result. The function next copies the first string into the new space and then calls `strcat` to concatenate the second string.

```
char *concat(const char *s1, const char *s2)
{
  char *result;

  result = malloc(strlen(s1) + strlen(s2) + 1);
  if (result == NULL) {
    printf("Error: malloc failed in concat\n");
    exit(EXIT_FAILURE);
  }
  strcpy(result, s1);
  strcat(result, s2);
  return result;
}
```

If `malloc` returns a null pointer, `concat` prints an error message and terminates the program. That's not always the right action to take; some programs need to recover from memory allocation failures and continue running.

Here's how the `concat` function might be called:

```
p = concat("abc", "def");
```

After the call, `p` will point to the string `"abcdef"`, which is stored in a dynamically allocated array. The array is seven characters long, including the null character at the end.

free function ➤17.4

Functions such as `concat` that dynamically allocate storage must be used with care. When the string that `concat` returns is no longer needed, we'll want to call the `free` function to release the space that the string occupies. If we don't, the program may eventually run out of memory.

Arrays of Dynamically Allocated Strings

In Section 13.7, we tackled the problem of storing strings in an array. We found that storing strings as rows in a two-dimensional array of characters can waste space, so we tried setting up an array of pointers to string literals. The techniques of Section 13.7 work just as well if the elements of an array are pointers to dynamically allocated strings. To illustrate this point, let's rewrite the `remind.c` program of Section 13.5, which prints a one-month list of daily reminders.

PROGRAM

Printing a One-Month Reminder List (Revisited)

The original `remind.c` program stores the reminder strings in a two-dimensional array of characters, with each row of the array containing one string. After the program reads a day and its associated reminder, it searches the array to determine where the day belongs, using `strcmp` to do comparisons. It then uses `strcpy` to move all strings below that point down one position. Finally, the program copies the day into the array and calls `strcat` to append the reminder to the day.

In the new program (`remind2.c`), the array will be one-dimensional; its elements will be pointers to dynamically allocated strings. Switching to dynamically allocated strings in this program will have two primary advantages. First, we can use space more efficiently by allocating the exact number of characters needed to store a reminder, rather than storing the reminder in a fixed number of characters as the original program does. Second, we won't need to call `strcpy` to move existing reminder strings in order to make room for a new reminder. Instead, we'll merely move *pointers* to strings.

Here's the new program, with changes in **bold.** Switching from a two-dimensional array to an array of pointers turns out to be remarkably easy: we'll only need to change eight lines of the program.

remind2.c

```
/* Prints a one-month reminder list (dynamic string version) */

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

#define MAX_REMIND 50   /* maximum number of reminders */
#define MSG_LEN 60      /* max length of reminder message */

int read_line(char str[], int n);
```

```
int main(void)
{
  char *reminders[MAX_REMIND];
  char day_str[3], msg_str[MSG_LEN+1];
  int day, i, j, num_remind = 0;

  for (;;) {
    if (num_remind == MAX_REMIND) {
      printf("-- No space left --\n");
      break;
    }

    printf("Enter day and reminder: ");
    scanf("%2d", &day);
    if (day == 0)
      break;
    sprintf(day_str, "%2d", day);
    read_line(msg_str, MSG_LEN);

    for (i = 0; i < num_remind; i++)
      if (strcmp(day_str, reminders[i]) < 0)
        break;
    for (j = num_remind; j > i; j--)
      reminders[j] = reminders[j-1];

    reminders[i] = malloc(2 + strlen(msg_str) + 1);
    if (reminders[i] == NULL) {
      printf("-- No space left --\n");
      break;
    }

    strcpy(reminders[i], day_str);
    strcat(reminders[i], msg_str);

    num_remind++;
  }

  printf("\nDay Reminder\n");
  for (i = 0; i < num_remind; i++)
    printf(" %s\n", reminders[i]);

  return 0;
}

int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';
  return i;
}
```

17.3 Dynamically Allocated Arrays

Dynamically allocated arrays have the same advantages as dynamically allocated strings (not surprisingly, since strings *are* arrays). When we're writing a program, it's often difficult to estimate the proper size for an array; it would be more convenient to wait until the program is run to decide how large the array should be. C solves this problem by allowing a program to allocate space for an array during execution, then access the array through a pointer to its first element. The close relationship between arrays and pointers, which we explored in Chapter 12, makes a dynamically allocated array just as easy to use as an ordinary array.

Although `malloc` can allocate space for an array, the `calloc` function is sometimes used instead, since it initializes the memory that it allocates. The `realloc` function allows us to make an array "grow" or "shrink" as needed.

Using `malloc` to Allocate Storage for an Array

We can use `malloc` to allocate space for an array in much the same way we used it to allocate space for a string. The primary difference is that the elements of an arbitrary array won't necessarily be one byte long, as they are in a string. As a result, we'll need to use the `sizeof` operator to calculate the amount of space required for each element.

`sizeof` operator ➤7.6

Suppose we're writing a program that needs an array of `n` integers, where `n` is to be computed during the execution of the program. We'll first declare a pointer variable:

```
int *a;
```

Once the value of `n` is known, we'll have the program call `malloc` to allocate space for the array:

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

Always use `sizeof` when calculating how much space is needed for an array. Failing to allocate enough memory can have severe consequences. Consider the following attempt to allocate space for an array of `n` integers:

```
a = malloc(n * 2);
```

If `int` values are larger than two bytes (as they are on most computers), `malloc` won't allocate a large enough block of memory. When we later try to access elements of the array, the program may crash or behave erratically.

Once it points to a dynamically allocated block of memory, we can ignore the fact that `a` is a pointer and use it instead as an array name, thanks to the relation-

ship between arrays and pointers in C. For example, we could use the following loop to initialize the array that a points to:

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

We also have the option of using pointer arithmetic instead of subscripting to access the elements of the array.

The calloc Function

Although the malloc function can be used to allocate memory for an array, C provides an alternative—the calloc function—that's sometimes better. calloc has the following prototype in <stdlib.h>:

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

calloc allocates space for an array with nmemb elements, each of which is size bytes long; it returns a null pointer if the requested space isn't available. After allocating the memory, calloc initializes it by setting all bits to 0. For example, the following call of calloc allocates space for an array of n integers, which are all guaranteed to be zero initially:

Q&A

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

Since calloc clears the memory that it allocates but malloc doesn't, we may occasionally want to use calloc to allocate space for an object other than an array. By calling calloc with 1 as its first argument, we can allocate space for a data item of any type:

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

p = calloc(1, sizeof(struct point));
```

After this statement has been executed, p will point to a structure whose x and y members have been set to zero.

The realloc Function

Once we've allocated memory for an array, we may later find that it's too large or too small. The realloc function can resize the array to better suit our needs. The following prototype for realloc appears in <stdlib.h>:

```
void *realloc(void *ptr, size_t size);
```

When realloc is called, ptr must point to a memory block obtained by a previous call of malloc, calloc, or realloc. The size parameter represents the new size of the block, which may be larger or smaller than the original size. Although realloc doesn't require that ptr point to memory that's being used as an array, in practice it usually does.

Be sure that a pointer passed to `realloc` came from a previous call of `malloc`, `calloc`, or `realloc`. If it didn't, calling `realloc` causes undefined behavior.

The C standard spells out a number of rules concerning the behavior of `realloc`:

- When it expands a memory block, `realloc` doesn't initialize the bytes that are added to the block.
- If `realloc` can't enlarge the memory block as requested, it returns a null pointer; the data in the old memory block is unchanged.
- If `realloc` is called with a null pointer as its first argument, it behaves like `malloc`.
- If `realloc` is called with `0` as its second argument, it frees the memory block.

The C standard stops short of specifying exactly how `realloc` works. Still, we expect it to be reasonably efficient. When asked to reduce the size of a memory block, `realloc` should shrink the block "in place," without moving the data stored in the block. By the same token, `realloc` should always attempt to expand a memory block without moving it. If it's unable to enlarge the block (because the bytes following the block are already in use for some other purpose), `realloc` will allocate a new block elsewhere, then copy the contents of the old block into the new one.

Once `realloc` has returned, be sure to update all pointers to the memory block, since it's possible that `realloc` has moved the block elsewhere.

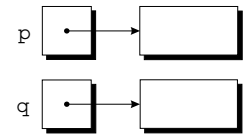
17.4 Deallocating Storage

`malloc` and the other memory allocation functions obtain memory blocks from a storage pool known as the ***heap.*** Calling these functions too often—or asking them for large blocks of memory—can exhaust the heap, causing the functions to return a null pointer.

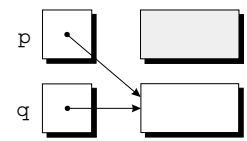
To make matters worse, a program may allocate blocks of memory and then lose track of them, thereby wasting space. Consider the following example:

```
p = malloc(…);
q = malloc(…);
p = q;
```

After the first two statements have been executed, p points to one memory block, while q points to another:



After q is assigned to p, both variables now point to the second memory block:



There are no pointers to the first block (shaded), so we'll never be able to use it again.

A block of memory that's no longer accessible to a program is said to be ***garbage.*** A program that leaves garbage behind has a ***memory leak.*** Some languages provide a ***garbage collector*** that automatically locates and recycles garbage, but C doesn't. Instead, each C program is responsible for recycling its own garbage by calling the `free` function to release unneeded memory.

The `free` Function

The `free` function has the following prototype in `<stdlib.h>`:

```
void free(void *ptr);
```

Using `free` is easy; we simply pass it a pointer to a memory block that we no longer need:

```
p = malloc(…);
q = malloc(…);
free(p);
p = q;
```

Calling `free` releases the block of memory that `p` points to. This block is now available for reuse in subsequent calls of `malloc` or other memory allocation functions.

The argument to `free` must be a pointer that was previously returned by a memory allocation function. (The argument may also be a null pointer, in which case the call of `free` has no effect.) Passing `free` a pointer to any other object (such as a variable or array element) causes undefined behavior.

The "Dangling Pointer" Problem

Although the `free` function allows us to reclaim memory that's no longer needed, using it leads to a new problem: ***dangling pointers.*** The call `free(p)` deallocates the memory block that `p` points to, but doesn't change `p` itself. If we forget that `p` no longer points to a valid memory block, chaos may ensue:

```
char *p = malloc(4);
…
free(p);
…
strcpy(p, "abc");    /*** WRONG ***/
```

Modifying the memory that `p` points to is a serious error, since our program no longer has control of that memory.

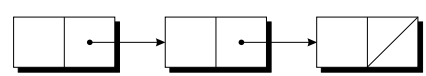
Attempting to access or modify a deallocated memory block causes undefined behavior. Trying to modify a deallocated memory block is likely to have disastrous consequences that may include a program crash.

Dangling pointers can be hard to spot, since several pointers may point to the same block of memory. When the block is freed, all the pointers are left dangling.

17.5 Linked Lists

Dynamic storage allocation is especially useful for building lists, trees, graphs, and other linked data structures. We'll look at linked lists in this section; a discussion of other linked data structures is beyond the scope of this book. For more information, consult a book such as Robert Sedgewick's *Algorithms in C, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching,* Third Edition (Reading, Mass.: Addison-Wesley, 1998).

A ***linked list*** consists of a chain of structures (called ***nodes***), with each node containing a pointer to the next node in the chain:



The last node in the list contains a null pointer, shown here as a diagonal line.

In previous chapters, we've used an array whenever we've needed to store a collection of data items; linked lists give us an alternative. A linked list is more flexible than an array: we can easily insert and delete nodes in a linked list, allowing the list to grow and shrink as needed. On the other hand, we lose the "random access" capability of an array. Any element of an array can be accessed in the same

amount of time; accessing a node in a linked list is fast if the node is close to the beginning of the list, slow if it's near the end.

This section describes how to set up a linked list in C. It also shows how to perform several common operations on linked lists: inserting a node at the beginning of a list, searching for a node, and deleting a node.

Declaring a Node Type

To set up a linked list, the first thing we'll need is a structure that represents a single node in the list. For simplicity, let's assume that a node contains nothing but an integer (the node's data) plus a pointer to the next node in the list. Here's what our node structure will look like:

```
struct node {
  int value;            /* data stored in the node  */
  struct node *next;    /* pointer to the next node */
};
```

Notice that the `next` member has type `struct node *`, which means that it can store a pointer to a `node` structure. There's nothing special about the name `node`, by the way; it's just an ordinary structure tag.

One aspect of the `node` structure deserves special mention. As Section 16.2 explained, we normally have the option of using either a tag or a `typedef` name to define a name for a particular kind of structure. However, when a structure has a member that points to the same kind of structure, as `node` does, we're required to use a structure tag. Without the `node` tag, we'd have no way to declare the type of `next`.

Q&A

Now that we have the `node` structure declared, we'll need a way to keep track of where the list begins. In other words, we'll need a variable that always points to the first node in the list. Let's name the variable `first`:

```
struct node *first = NULL;
```

Setting `first` to `NULL` indicates that the list is initially empty.

Creating a Node

As we construct a linked list, we'll want to create nodes one by one, adding each to the list. Creating a node requires three steps:

1. Allocate memory for the node.
2. Store data in the node.
3. Insert the node into the list.

We'll concentrate on the first two steps for now.

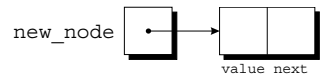
When we create a node, we'll need a variable that can point to the node temporarily, until it's been inserted into the list. Let's call this variable `new_node`:

```
struct node *new_node;
```

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 ***/
```

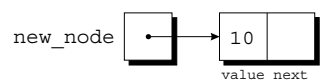
The program will still compile, but `malloc` will allocate only enough memory for a *pointer* to a `node` structure. The likely result is a crash later, when the program attempts to store data in the node that `new_node` is presumably pointing to.

Q&A

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:

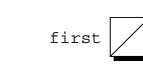
```
new_node->next = first;
```

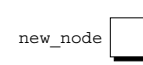
Second, we'll make `first` point to the new node:

```
first = new_node;
```

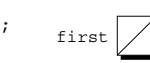
Will these statements work if the list is empty when we insert a node? Yes, fortunately. To make sure this is true, let's trace the process of inserting two nodes into an empty list. We'll insert a node containing the number 10 first, followed by a node containing 20. In the figures that follow, null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

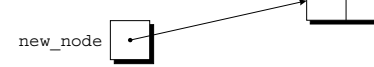
```
first = NULL;
```



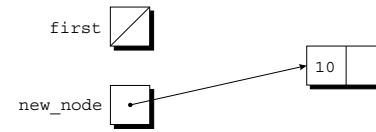



```
new_node = malloc(sizeof(struct node));
```

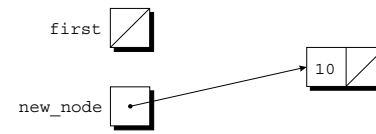




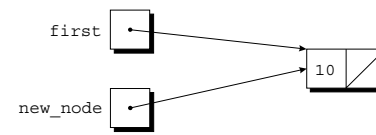
```
new_node->value = 10;
```



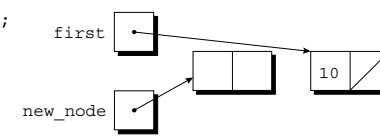

```
new_node->next = first;
```



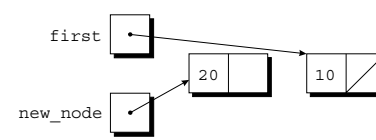

```
first = new_node;
```



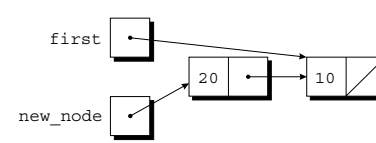

```
new_node = malloc(sizeof(struct node));
```



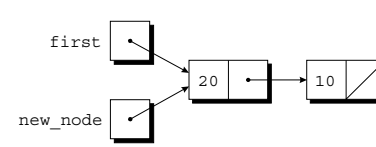

```
new_node->value = 20;
```




```
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.

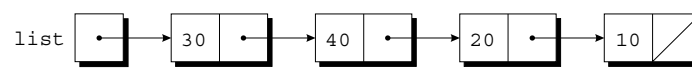
Step 1 is harder than it looks. If we search the list in the obvious way, we'll end up with a pointer to the node to be deleted. Unfortunately, we won't be able to perform step 2, which requires changing the *previous* node.

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:

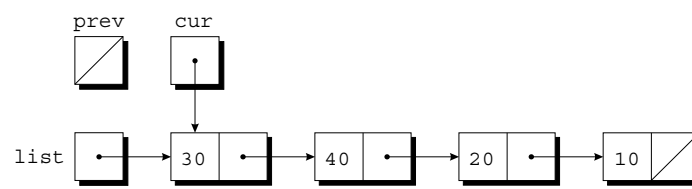
```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next)
  ;
```

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).

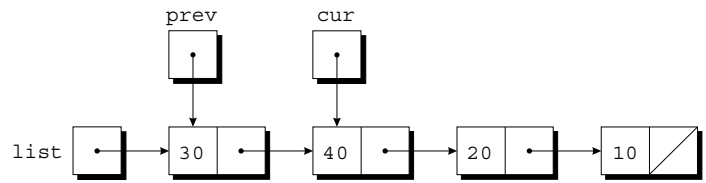
To see how this loop works, let's assume that `list` points to a list containing 30, 40, 20, and 10, in that order:



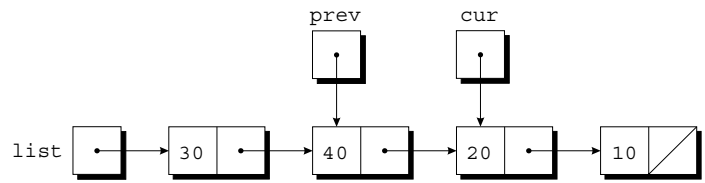
Let's say that `n` is 20, so our goal is to delete the third node in the list. After `cur = list, prev = NULL` has been executed, `cur` points to the first node in the list:



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:

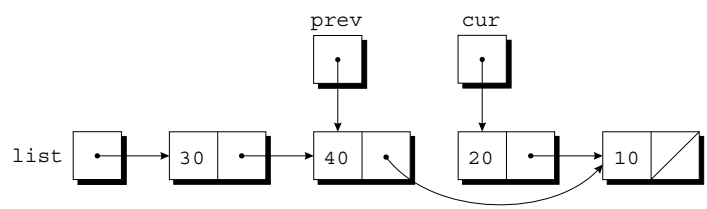


Since `cur` now points to the node containing 20, the condition `cur->value != n` is false and the loop terminates.

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 π/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

Q&A

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.

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

2. (W) 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.