10 Program Organization

As Will Rogers would have said, "There is no such thing as a free variable."

Having covered functions in Chapter 9, we're ready to confront several issues that arise when a program contains more than one function. The chapter begins with a discussion of the differences between local variables (Section 10.1) and external variables (Section 10.2). Section 10.3 then considers blocks (compound statements containing declarations). Section 10.4 tackles the scope rules that apply to local names, external names, and names declared in blocks. Finally, Section 10.5 suggests a way to organize function prototypes, function definitions, variable declarations, and the other parts of a C program.

10.1 Local Variables

A variable declared in the body of a function is said to be ***local*** to the function. In the following function, `sum` is a local variable:

```
int sum_digits(int n)
{
  int sum = 0;   /* local variable */

  while (n > 0) {
    sum += n % 10;
    n /= 10;
  }

  return sum;
}
```

By default, local variables have the following properties:

- ***Automatic storage duration.*** The ***storage duration*** (or ***extent***) of a variable is the portion of program execution during which storage for the variable exists. Storage for a local variable is "automatically" allocated when the enclosing function is called and deallocated when the function returns, so the variable is said to have ***automatic storage duration.*** A local variable doesn't retain its value when its enclosing function returns. When the function is called again, there's no guarantee that the variable will still have its old value.
- ***Block scope.*** The ***scope*** of a variable is the portion of the program text in which the variable can be referenced. A local variable has ***block scope:*** it is visible from its point of declaration to the end of the enclosing function body. Since the scope of a local variable doesn't extend beyond the function to which it belongs, other functions can use the same name for other purposes.

Section 18.2 covers these and other related concepts in more detail.

C99 Since C99 doesn't require variable declarations to come at the beginning of a function, it's possible for a local variable to have a very small scope. In the following example, the scope of `i` doesn't begin until the line on which it's declared, which could be near the end of the function body:

```
void f(void)
{
  ...
  int i; ──┐
  ...      ├─ scope of i
}        ──┘
```

Static Local Variables

Putting the word `static` in the declaration of a local variable causes it to have ***static storage duration*** instead of automatic storage duration. A variable with static storage duration has a permanent storage location, so it retains its value throughout the execution of the program. Consider the following function:

```
void f(void)
{
  static int i;   /* static local variable */
  ...
}
```

Since the local variable `i` has been declared `static`, it occupies the same memory location throughout the execution of the program. When `f` returns, `i` won't lose its value. Q&A

A static local variable still has block scope, so it's not visible to other functions. In a nutshell, a static variable is a place to hide data from other functions but retain it for future calls of the same function.

Parameters

Parameters have the same properties—automatic storage duration and block scope—as local variables. In fact, the only real difference between parameters and local variables is that each parameter is initialized automatically when a function is called (by being assigned the value of the corresponding argument).

10.2 External Variables

Passing arguments is one way to transmit information to a function. Functions can also communicate through ***external variables***—variables that are declared outside the body of any function.

The properties of external variables (or ***global variables,*** as they're sometimes called) are different from those of local variables:

- ***Static storage duration.*** External variables have static storage duration, just like local variables that have been declared `static`. A value stored in an external variable will stay there indefinitely.
- ***File scope.*** An external variable has ***file scope:*** it is visible from its point of declaration to the end of the enclosing file. As a result, an external variable can be accessed (and potentially modified) by all functions that follow its declaration.

Example: Using External Variables to Implement a Stack

To illustrate how external variables might be used, let's look at a data structure known as a ***stack***. (Stacks are an abstract concept, not a C feature; they can be implemented in most programming languages.) A stack, like an array, can store multiple data items of the same type. However, the operations on a stack are limited: we can either ***push*** an item onto the stack (add it to one end—the "stack top") or ***pop*** it from the stack (remove it from the same end). Examining or modifying an item that's not at the top of the stack is forbidden.

One way to implement a stack in C is to store its items in an array, which we'll call `contents`. A separate integer variable named `top` marks the position of the stack top. When the stack is empty, `top` has the value 0. To push an item on the stack, we simply store the item in `contents` at the position indicated by `top`, then increment `top`. Popping an item requires decrementing `top`, then using it as an index into `contents` to fetch the item that's being popped.

Based on this outline, here's a program fragment (not a complete program) that declares the `contents` and `top` variables for a stack and provides a set of functions that represent operations on the stack. All five functions need access to the `top` variable, and two functions need access to `contents`, so we'll make `contents` and `top` external.

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

#define STACK_SIZE 100

/* external variables */
int contents[STACK_SIZE];
int top = 0;

void make_empty(void)
{
  top = 0;
}

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

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

void push(int i)
{
  if (is_full())
    stack_overflow();
  else
    contents[top++] = i;
}

int pop(void)
{
  if (is_empty())
    stack_underflow();
  else
    return contents[--top];
}
```

Pros and Cons of External Variables

External variables are convenient when many functions must share a variable or when a few functions share a large number of variables. In most cases, however, it's better for functions to communicate through parameters rather than by sharing variables. Here's why:

- If we change an external variable during program maintenance (by altering its type, say), we'll need to check every function in the same file to see how the change affects it.

- If an external variable is assigned an incorrect value, it may be difficult to identify the guilty function. It's like trying to solve a murder committed at a crowded party—there's no easy way to narrow the list of suspects.
- Functions that rely on external variables are hard to reuse in other programs. A function that depends on external variables isn't self-contained; to reuse the function, we'll have to drag along any external variables that it needs.

Many C programmers rely far too much on external variables. One common abuse: using the same external variable for different purposes in different functions. Suppose that several functions need a variable named `i` to control a `for` statement. Instead of declaring `i` in each function that uses it, some programmers declare it at the top of the program, thereby making the variable visible to all functions. This practice is poor not only for the reasons listed earlier, but also because it's misleading; someone reading the program later may think that the uses of the variable are related, when in fact they're not.

When you use external variables, make sure they have meaningful names. (Local variables don't always need meaningful names: it's often hard to think of a better name than `i` for the control variable in a `for` loop.) If you find yourself using names like `i` and `temp` for external variables, that's a clue that perhaps they should really be local variables.

⚠ Making variables external when they should be local can lead to some rather frustrating bugs. Consider the following example, which is supposed to display a 10 × 10 arrangement of asterisks:

```
int i;

void print_one_row(void)
{
  for (i = 1; i <= 10; i++)
    printf("*");
}

void print_all_rows(void)
{
  for (i = 1; i <= 10; i++) {
    print_one_row();
    printf("\n");
  }
}
```

Instead of printing 10 rows, `print_all_rows` prints only one row. When `print_one_row` returns after being called the first time, `i` will have the value 11. The `for` statement in `print_all_rows` then increments `i` and tests whether it's less than or equal to 10. It's not, so the loop terminates and the function returns.

PROGRAM Guessing a Number

To get more experience with external variables, we'll write a simple game-playing program. The program generates a random number between 1 and 100, which the user attempts to guess in as few tries as possible. Here's what the user will see when the program is run:

```
Guess the secret number between 1 and 100.

A new number has been chosen.
Enter guess: 55
Too low; try again.
Enter guess: 65
Too high; try again.
Enter guess: 60
Too high; try again.
Enter guess: 58
You won in 4 guesses!

Play again? (Y/N) y

A new number has been chosen.
Enter guess: 78
Too high; try again.
Enter guess: 34
You won in 2 guesses!

Play again? (Y/N) n
```

This program will need to carry out several different tasks: initializing the random number generator, choosing a secret number, and interacting with the user until the correct number is picked. If we write a separate function to handle each task, we might end up with the following program.

guess.c

```
/* Asks user to guess a hidden number */

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

#define MAX_NUMBER 100

/* external variable */
int secret_number;

/* prototypes */
void initialize_number_generator(void);
void choose_new_secret_number(void);
void read_guesses(void);

int main(void)
{
  char command;
  printf("Guess the secret number between 1 and %d.\n\n",
         MAX_NUMBER);
  initialize_number_generator();
  do {
    choose_new_secret_number();
    printf("A new number has been chosen.\n");
    read_guesses();
    printf("Play again? (Y/N) ");
    scanf(" %c", &command);
    printf("\n");
  } while (command == 'y' || command == 'Y');

  return 0;
}

/**********************************************************
 * initialize_number_generator: Initializes the random    *
 *                              number generator using    *
 *                              the time of day.          *
 **********************************************************/
void initialize_number_generator(void)
{
  srand((unsigned) time(NULL));
}

/**********************************************************
 * choose_new_secret_number: Randomly selects a number    *
 *                           between 1 and MAX_NUMBER and *
 *                           stores it in secret_number.  *
 **********************************************************/
void choose_new_secret_number(void)
{
  secret_number = rand() % MAX_NUMBER + 1;
}

/**********************************************************
 * read_guesses: Repeatedly reads user guesses and tells  *
 *               the user whether each guess is too low,  *
 *               too high, or correct. When the guess is  *
 *               correct, prints the total number of      *
 *               guesses and returns.                     *
 **********************************************************/
void read_guesses(void)
{
  int guess, num_guesses = 0;

  for (;;) {
    num_guesses++;
    printf("Enter guess: ");
    scanf("%d", &guess);
    if (guess == secret_number) {
      printf("You won in %d guesses!\n\n", num_guesses);
      return;
    } else if (guess < secret_number)
```

```
      printf("Too low; try again.\n");
    else
      printf("Too high; try again.\n");
  }
}
```

time function ➤26.3
srand function ➤26.2
rand function ➤26.2

For random number generation, the guess.c program relies on the time, srand, and rand functions, which we first used in deal.c (Section 8.2). This time, we're scaling the return value of rand so that it falls between 1 and MAX_NUMBER.

Although guess.c works fine, it relies on an external variable. We made secret_number external so that both choose_new_secret_number and read_guesses could access it. If we alter choose_new_secret_number and read_guesses just a little, we should be able to move secret_number into the main function. We'll modify choose_new_secret_number so that it returns the new number, and we'll rewrite read_guesses so that secret_number can be passed to it as an argument.

Here's our new program, with changes in **bold**:

guess2.c

```
/* Asks user to guess a hidden number */

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

#define MAX_NUMBER 100

/* prototypes */
void initialize_number_generator(void);
int new_secret_number(void);
void read_guesses(int secret_number);

int main(void)
{
  char command;
  int secret_number;

  printf("Guess the secret number between 1 and %d.\n\n",
         MAX_NUMBER);
  initialize_number_generator();
  do {
    secret_number = new_secret_number();
    printf("A new number has been chosen.\n");
    read_guesses(secret_number);
    printf("Play again? (Y/N) ");
    scanf(" %c", &command);
    printf("\n");
  } while (command == 'y' || command == 'Y');

  return 0;
}
```

```
/**********************************************************
 * initialize_number_generator: Initializes the random    *
 *                              number generator using    *
 *                              the time of day.          *
 **********************************************************/
void initialize_number_generator(void)
{
  srand((unsigned) time(NULL));
}

/**********************************************************
 * new_secret_number: Returns a randomly chosen number    *
 *                    between 1 and MAX_NUMBER.           *
 **********************************************************/
int new_secret_number(void)
{
  return rand() % MAX_NUMBER + 1;
}

/**********************************************************
 * read_guesses: Repeatedly reads user guesses and tells  *
 *               the user whether each guess is too low,  *
 *               too high, or correct. When the guess is  *
 *               correct, prints the total number of      *
 *               guesses and returns.                     *
 **********************************************************/
void read_guesses(int secret_number)
{
  int guess, num_guesses = 0;

  for (;;) {
    num_guesses++;
    printf("Enter guess: ");
    scanf("%d", &guess);
    if (guess == secret_number) {
      printf("You won in %d guesses!\n\n", num_guesses);
      return;
    } else if (guess < secret_number)
      printf("Too low; try again.\n");
    else
      printf("Too high; try again.\n");
  }
}
```

10.3 Blocks

In Section 5.2, we encountered compound statements of the form

{ *statements* }

It turns out that C allows compound statements to contain declarations as well:

block { *declarations statements* }

I'll use the term ***block*** to describe such a compound statement. Here's an example of a block:

```
if (i > j) {
  /* swap values of i and j */
  int temp = i;
  i = j;
  j = temp;
}
```

By default, the storage duration of a variable declared in a block is automatic: storage for the variable is allocated when the block is entered and deallocated when the block is exited. The variable has block scope; it can't be referenced outside the block. A variable that belongs to a block can be declared `static` to give it static storage duration.

The body of a function is a block. Blocks are also useful inside a function body when we need variables for temporary use. In our last example, we needed a variable temporarily so that we could swap the values of `i` and `j`. Putting temporary variables in blocks has two advantages: (1) It avoids cluttering the declarations at the beginning of the function body with variables that are used only briefly. (2) It reduces name conflicts. In our example, the name `temp` can be used elsewhere in the same function for different purposes—the `temp` variable is strictly local to the block in which it's declared.

C99 C99 allows variables to be declared anywhere within a block, just as it allows variables to be declared anywhere within a function.

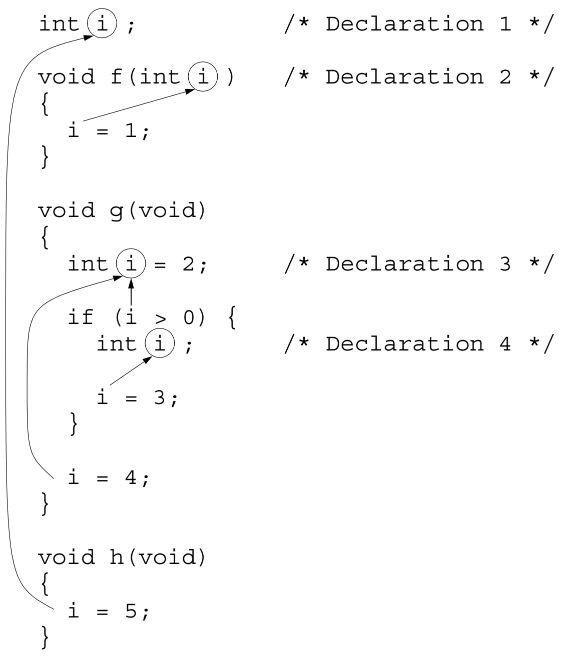
10.4 Scope

In a C program, the same identifier may have several different meanings. C's scope rules enable the programmer (and the compiler) to determine which meaning is relevant at a given point in the program.

Here's the most important scope rule: When a declaration inside a block names an identifier that's already visible (because it has file scope or because it's declared in an enclosing block), the new declaration temporarily "hides" the old one, and the identifier takes on a new meaning. At the end of the block, the identifier regains its old meaning.

Consider the (somewhat extreme) example at the top of the next page, in which the identifier `i` has four different meanings:

- In Declaration 1, `i` is a variable with static storage duration and file scope.



```
int i ;              /* Declaration 1 */

void f(int i )       /* Declaration 2 */
{
  i = 1;
}

void g(void)
{
  int i = 2;         /* Declaration 3 */

  if (i > 0) {
    int i ;          /* Declaration 4 */

    i = 3;
  }

  i = 4;
}

void h(void)
{
  i = 5;
}
```

- In Declaration 2, `i` is a parameter with block scope.
- In Declaration 3, `i` is an automatic variable with block scope.
- In Declaration 4, `i` is also automatic and has block scope.

`i` is used five times. C's scope rules allow us to determine the meaning of `i` in each case:

- The `i = 1` assignment refers to the parameter in Declaration 2, not the variable in Declaration 1, since Declaration 2 hides Declaration 1.
- The `i > 0` test refers to the variable in Declaration 3, since Declaration 3 hides Declaration 1 and Declaration 2 is out of scope.
- The `i = 3` assignment refers to the variable in Declaration 4, which hides Declaration 3.
- The `i = 4` assignment refers to the variable in Declaration 3. It can't refer to Declaration 4, which is out of scope.
- The `i = 5` assignment refers to the variable in Declaration 1.

10.5 Organizing a C Program

Now that we've seen the major elements that make up a C program, it's time to develop a strategy for their arrangement. For now, we'll assume that a program always fits into a single file. Chapter 15 shows how to organize a program that's split over several files.

So far, we've seen that a program may contain the following:

Preprocessing directives such as `#include` and `#define`
Type definitions
Declarations of external variables
Function prototypes
Function definitions

C imposes only a few rules on the order of these items: A preprocessing directive doesn't take effect until the line on which it appears. A type name can't be used until it's been defined. A variable can't be used until it's declared. Although C isn't as picky about functions, I strongly recommend that every function be defined or declared prior to its first call. (C99 makes this a requirement anyway.)

C99

There are several ways to organize a program so that these rules are obeyed. Here's one possible ordering:

`#include` directives
`#define` directives
Type definitions
Declarations of external variables
Prototypes for functions other than `main`
Definition of `main`
Definitions of other functions

It makes sense to put `#include` directives first, since they bring in information that will likely be needed in several places within the program. `#define` directives create macros, which are generally used throughout the program. Putting type definitions above the declarations of external variables is logical, since the declarations of these variables may refer to the type names just defined. Declaring external variables next makes them available to all the functions that follow. Declaring all functions except for `main` avoids the problems that arise when a function is called before the compiler has seen its prototype. This practice also makes it possible to arrange the function definitions in any order whatsoever: alphabetically by function name or with related functions grouped together, for example. Defining `main` before the other functions makes it easier for a reader to locate the program's starting point.

A final suggestion: Precede each function definition by a boxed comment that gives the name of the function, explains its purpose, discusses the meaning of each parameter, describes its return value (if any), and lists any side effects it has (such as modifying external variables).

PROGRAM Classifying a Poker Hand

To show how a C program might be organized, let's attempt a program that's a little more complex than our previous examples. The program will read and classify a poker hand. Each card in the hand will have both a *suit* (clubs, diamonds, hearts, or spades) and a *rank* (two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, or ace). We won't allow the use of jokers, and we'll assume that aces are high. The program will read a hand of five cards, then classify the hand into one of the following categories (listed in order from best to worst):

straight flush (both a straight and a flush)
four-of-a-kind (four cards of the same rank)
full house (a three-of-a-kind and a pair)
flush (five cards of the same suit)
straight (five cards with consecutive ranks)
three-of-a-kind (three cards of the same rank)
two pairs
pair (two cards of the same rank)
high card (any other hand)

If a hand falls into two or more categories, the program will choose the best one.

For input purposes, we'll abbreviate ranks and suits as follows (letters may be either upper- or lower-case):

Ranks: `2 3 4 5 6 7 8 9 t j q k a`
Suits: `c d h s`

If the user enters an illegal card or tries to enter the same card twice, the program will ignore the card, issue an error message, and then request another card. Entering the number 0 instead of a card will cause the program to terminate.

A session with the program will have the following appearance:

```
Enter a card: 2s
Enter a card: 5s
Enter a card: 4s
Enter a card: 3s
Enter a card: 6s
Straight flush

Enter a card: 8c
Enter a card: as
Enter a card: 8c
Duplicate card; ignored.
Enter a card: 7c
Enter a card: ad
Enter a card: 3h
Pair

Enter a card: 6s
Enter a card: d2
Bad card; ignored.
Enter a card: 2d
Enter a card: 9c
Enter a card: 4h
Enter a card: ts
```

```
High card

Enter a card: 0
```

From this description of the program, we see that it has three tasks:

Read a hand of five cards.
Analyze the hand for pairs, straights, and so forth.
Print the classification of the hand.

We'll divide the program into three functions—`read_cards`, `analyze_hand`, and `print_result`—that perform these three tasks. `main` does nothing but call these functions inside an endless loop. The functions will need to share a fairly large amount of information, so we'll have them communicate through external variables. `read_cards` will store information about the hand into several external variables. `analyze_hand` will then examine these variables, storing its findings into other external variables for the benefit of `print_result`.

Based on this preliminary design, we can begin to sketch an outline of the program:

```
/* #include directives go here */

/* #define directives go here */

/* declarations of external variables go here */

/* prototypes */
void read_cards(void);
void analyze_hand(void);
void print_result(void);

/**********************************************************
 * main: Calls read_cards, analyze_hand, and print_result *
 *       repeatedly.                                      *
 **********************************************************/
int main(void)
{
  for (;;) {
    read_cards();
    analyze_hand();
    print_result();
  }
}

/**********************************************************
 * read_cards:  Reads the cards into external variables;  *
 *              checks for bad cards and duplicate cards. *
 **********************************************************/
void read_cards(void)
{
  ...
}
```

```
/**********************************************************
 * analyze_hand: Determines whether the hand contains a   *
 *               straight, a flush, four-of-a-kind,       *
 *               and/or three-of-a-kind; determines the   *
 *               number of pairs; stores the results into *
 *               external variables.                      *
 **********************************************************/
void analyze_hand(void)
{
  …
}

/**********************************************************
 * print_result: Notifies the user of the result, using   *
 *               the external variables set by            *
 *               analyze_hand.                            *
 **********************************************************/
void print_result(void)
{
  …
}
```

The most pressing question that remains is how to represent the hand of cards. Let's see what operations `read_cards` and `analyze_hand` will perform on the hand. During the analysis of the hand, `analyze_hand` will need to know how many cards are in each rank and each suit. This suggests that we use two arrays, `num_in_rank` and `num_in_suit`. The value of `num_in_rank[r]` will be the number of cards with rank `r`, and the value of `num_in_suit[s]` will be the number of cards with suit `s`. (We'll encode ranks as numbers between 0 and 12, and suits as numbers between 0 and 3.) We'll also need a third array, `card_exists`, so that `read_cards` can detect duplicate cards. Each time `read_cards` reads a card with rank `r` and suit `s`, it checks whether the value of `card_exists[r][s]` is `true`. If so, the card was previously entered; if not, `read_cards` assigns `true` to `card_exists[r][s]`.

Both the `read_cards` function and the `analyze_hand` function will need access to the `num_in_rank` and `num_in_suit` arrays, so I'll make them external variables. The `card_exists` array is used only by `read_cards`, so it can be local to that function. As a rule, variables should be made external only if necessary.

Having decided on the major data structures, we can now finish the program:

poker.c

```
/* Classifies a poker hand */

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

#define NUM_RANKS 13
#define NUM_SUITS 4
#define NUM_CARDS 5
```

```
/* external variables */
int num_in_rank[NUM_RANKS];
int num_in_suit[NUM_SUITS];
bool straight, flush, four, three;
int pairs;   /* can be 0, 1, or 2 */

/* prototypes */
void read_cards(void);
void analyze_hand(void);
void print_result(void);

/**********************************************************
 * main: Calls read_cards, analyze_hand, and print_result *
 *       repeatedly.                                      *
 **********************************************************/
int main(void)
{
  for (;;) {
    read_cards();
    analyze_hand();
    print_result();
  }
}

/**********************************************************
 * read_cards: Reads the cards into the external          *
 *             variables num_in_rank and num_in_suit;     *
 *             checks for bad cards and duplicate cards.  *
 **********************************************************/
void read_cards(void)
{
  bool card_exists[NUM_RANKS][NUM_SUITS];
  char ch, rank_ch, suit_ch;
  int rank, suit;
  bool bad_card;
  int cards_read = 0;

  for (rank = 0; rank < NUM_RANKS; rank++) {
    num_in_rank[rank] = 0;
    for (suit = 0; suit < NUM_SUITS; suit++)
      card_exists[rank][suit] = false;
  }

  for (suit = 0; suit < NUM_SUITS; suit++)
    num_in_suit[suit] = 0;

  while (cards_read < NUM_CARDS) {
    bad_card = false;

    printf("Enter a card: ");

    rank_ch = getchar();
    switch (rank_ch) {
      case '0':           exit(EXIT_SUCCESS);
      case '2':           rank = 0; break;
      case '3':           rank = 1; break;
      case '4':           rank = 2; break;
      case '5':           rank = 3; break;
      case '6':           rank = 4; break;
      case '7':           rank = 5; break;
      case '8':           rank = 6; break;
      case '9':           rank = 7; break;
      case 't': case 'T': rank = 8; break;
      case 'j': case 'J': rank = 9; break;
      case 'q': case 'Q': rank = 10; break;
      case 'k': case 'K': rank = 11; break;
      case 'a': case 'A': rank = 12; break;
      default:            bad_card = true;
    }

    suit_ch = getchar();
    switch (suit_ch) {
      case 'c': case 'C': suit = 0; break;
      case 'd': case 'D': suit = 1; break;
      case 'h': case 'H': suit = 2; break;
      case 's': case 'S': suit = 3; break;
      default:            bad_card = true;
    }

    while ((ch = getchar()) != '\n')
      if (ch != ' ') bad_card = true;

    if (bad_card)
      printf("Bad card; ignored.\n");
    else if (card_exists[rank][suit])
      printf("Duplicate card; ignored.\n");
    else {
      num_in_rank[rank]++;
      num_in_suit[suit]++;
      card_exists[rank][suit] = true;
      cards_read++;
    }
  }
}

/**********************************************************
 * analyze_hand: Determines whether the hand contains a   *
 *               straight, a flush, four-of-a-kind,       *
 *               and/or three-of-a-kind; determines the   *
 *               number of pairs; stores the results into *
 *               the external variables straight, flush,  *
 *               four, three, and pairs.                  *
 **********************************************************/
void analyze_hand(void)
{
  int num_consec = 0;
  int rank, suit;
  straight = false;
  flush = false;
  four = false;
  three = false;
  pairs = 0;

  /* check for flush */
  for (suit = 0; suit < NUM_SUITS; suit++)
    if (num_in_suit[suit] == NUM_CARDS)
      flush = true;

  /* check for straight */
  rank = 0;
  while (num_in_rank[rank] == 0) rank++;
  for (; rank < NUM_RANKS && num_in_rank[rank] > 0; rank++)
    num_consec++;
  if (num_consec == NUM_CARDS) {
    straight = true;
    return;
  }

  /* check for 4-of-a-kind, 3-of-a-kind, and pairs */
  for (rank = 0; rank < NUM_RANKS; rank++) {
    if (num_in_rank[rank] == 4) four = true;
    if (num_in_rank[rank] == 3) three = true;
    if (num_in_rank[rank] == 2) pairs++;
  }
}

/**********************************************************
 * print_result: Prints the classification of the hand,   *
 *               based on the values of the external      *
 *               variables straight, flush, four, three,  *
 *               and pairs.                               *
 **********************************************************/
void print_result(void)
{
  if (straight && flush) printf("Straight flush");
  else if (four)         printf("Four of a kind");
  else if (three &&
           pairs == 1)   printf("Full house");
  else if (flush)        printf("Flush");
  else if (straight)     printf("Straight");
  else if (three)        printf("Three of a kind");
  else if (pairs == 2)   printf("Two pairs");
  else if (pairs == 1)   printf("Pair");
  else                   printf("High card");

  printf("\n\n");
}
```

Notice the use of the `exit` function in `read_cards` (in case `'0'` of the first `switch` statement). `exit` is convenient for this program because of its ability to terminate execution from anywhere in the program.

Q & A

Q: What impact do local variables with static storage duration have on recursive functions? [p. 220]

A: When a function is called recursively, fresh copies are made of its automatic variables for each call. This doesn't occur for static variables, though. Instead, all calls of the function share the *same* static variables.

Q: In the following example, `j` is initialized to the same value as `i`, but there are two variables named `i`:

```
int i = 1;

void f(void)
{
  int j = i;
  int i = 2;
  …
}
```

Is this code legal? If so, what is `j`'s initial value, 1 or 2?

A: The code is indeed legal. The scope of a local variable doesn't begin until its declaration. Therefore, the declaration of `j` refers to the external variable named `i`. The initial value of `j` will be 1.

Exercises

Section 10.4

Ⓦ 1. The following program outline shows only function definitions and variable declarations.

```
int a;

void f(int b)
{
  int c;
}

void g(void)
{
  int d;
  {
    int e;
  }
}

int main(void)
{
  int f;
}
```

For each of the following scopes, list all variable and parameter names visible in that scope:

(a) The `f` function
(b) The `g` function
(c) The block in which `e` is declared
(d) The `main` function

2. The following program outline shows only function definitions and variable declarations.

```
int b, c;

void f(void)
{
  int b, d;
}

void g(int a)
{
  int c;
  {
    int a, d;
  }
}

int main(void)
{
  int c, d;
}
```

For each of the following scopes, list all variable and parameter names visible in that scope. If there's more than one variable or parameter with the same name, indicate which one is visible.

(a) The `f` function
(b) The `g` function
(c) The block in which `a` and `d` are declared
(d) The `main` function

*3. Suppose that a program has only one function (`main`). How many different variables named `i` could this program contain?

Programming Projects

1. Modify the stack example of Section 10.2 so that it stores characters instead of integers. Next, add a `main` function that asks the user to enter a series of parentheses and/or braces, then indicates whether or not they're properly nested:

```
Enter parentheses and/or braces: ((){}{()})
Parentheses/braces are nested properly
```

Hint: As the program reads characters, have it push each left parenthesis or left brace. When it reads a right parenthesis or brace, have it pop the stack and check that the item popped is a matching parenthesis or brace. (If not, the parentheses/braces aren't nested properly.) When the program reads the new-line character, have it check whether the stack is empty; if so, the parentheses/braces are matched. If the stack *isn't* empty (or if `stack_underflow` is ever called), the parentheses/braces aren't matched. If `stack_overflow` is called, have the program print the message `Stack overflow` and terminate immediately.

2. Modify the `poker.c` program of Section 10.5 by moving the `num_in_rank` and `num_in_suit` arrays into `main`, which will pass them as arguments to `read_cards` and `analyze_hand`.

Ⓦ 3. Remove the `num_in_rank`, `num_in_suit`, and `card_exists` arrays from the `poker.c` program of Section 10.5. Have the program store the cards in a 5 × 2 array instead. Each row of the array will represent a card. For example, if the array is named `hand`, then `hand[0][0]` will store the rank of the first card and `hand[0][1]` will store the suit of the first card.

4. Modify the `poker.c` program of Section 10.5 by having it recognize an additional category, "royal flush" (ace, king, queen, jack, ten of the same suit). A royal flush ranks higher than all other hands.

Ⓦ 5. Modify the `poker.c` program of Section 10.5 by allowing "ace-low" straights (ace, two, three, four, five).

6. Some calculators (notably those from Hewlett-Packard) use a system of writing mathematical expressions known as Reverse Polish Notation (RPN). In this notation, operators are placed *after* their operands instead of *between* their operands. For example, 1 + 2 would be written 1 2 + in RPN, and 1 + 2 * 3 would be written 1 2 3 * +. RPN expressions can easily be evaluated using a stack. The algorithm involves reading the operators and operands in an expression from left to right, performing the following actions:

   When an operand is encountered, push it onto the stack.

   When an operator is encountered, pop its operands from the stack, perform the operation on those operands, and then push the result onto the stack.

   Write a program that evaluates RPN expressions. The operands will be single-digit integers. The operators are +, –, *, /, and =. The = operator causes the top stack item to be displayed; afterwards, the stack is cleared and the user is prompted to enter another expression. The process continues until the user enters a character that is not an operator or operand:

```
Enter an RPN expression: 1 2 3 * + =
Value of expression: 7
Enter an RPN expression: 5 8 * 4 9 - / =
Value of expression: -8
Enter an RPN expression: q
```

   If the stack overflows, the program will display the message `Expression is too complex` and terminate. If the stack underflows (because of an expression such as 1 2 + +), the program will display the message `Not enough operands in expression` and terminate. *Hints:* Incorporate the stack code from Section 10.2 into your program. Use `scanf(" %c", &ch)` to read the operators and operands.

7. Write a program that prompts the user for a number and then displays the number, using characters to simulate the effect of a seven-segment display:

```
Enter a number: 491-9014

     _       _   _        
|_| |_|   | |_| | |   | |_|
  |  _|   |  _| |_|   |   |
```

   Characters other than digits should be ignored. Write the program so that the maximum number of digits is controlled by a macro named `MAX_DIGITS`, which has the value 10. If the number contains more than this number of digits, the extra digits are ignored. *Hints:* Use two external arrays. One is the `segments` array (see Exercise 6 in Chapter 8), which stores data representing the correspondence between digits and segments. The other array, `digits`, will be an array of characters with 4 rows (since each segmented digit is four characters high) and `MAX_DIGITS * 4` columns (digits are three characters wide, but a space is needed between digits for readability). Write your program as four functions: `main`, `clear_digits_array`, `process_digit`, and `print_digits_array`. Here are the prototypes for the latter three functions:

```
void clear_digits_array(void);
void process_digit(int digit, int position);
void print_digits_array(void);
```

`clear_digits_array` will store blank characters into all elements of the `digits` array. `process_digit` will store the seven-segment representation of `digit` into a specified position in the `digits` array (positions range from 0 to `MAX_DIGITS` – 1). `print_digits_array` will display the rows of the `digits` array, each on a single line, producing output such as that shown in the example.