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

W 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`.

W 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.

W 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