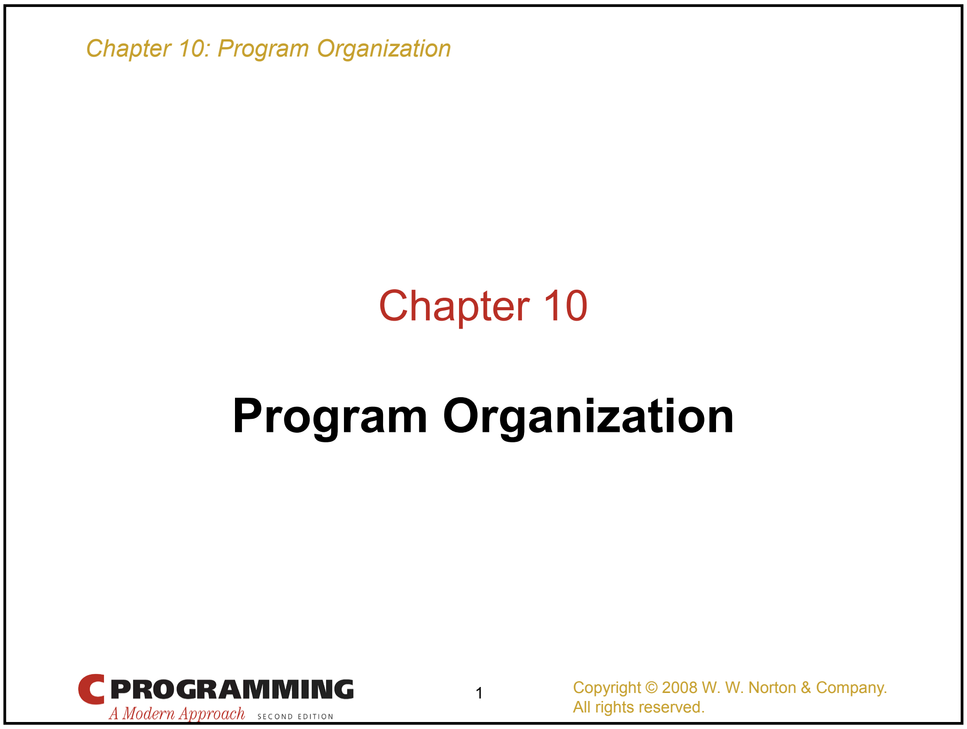
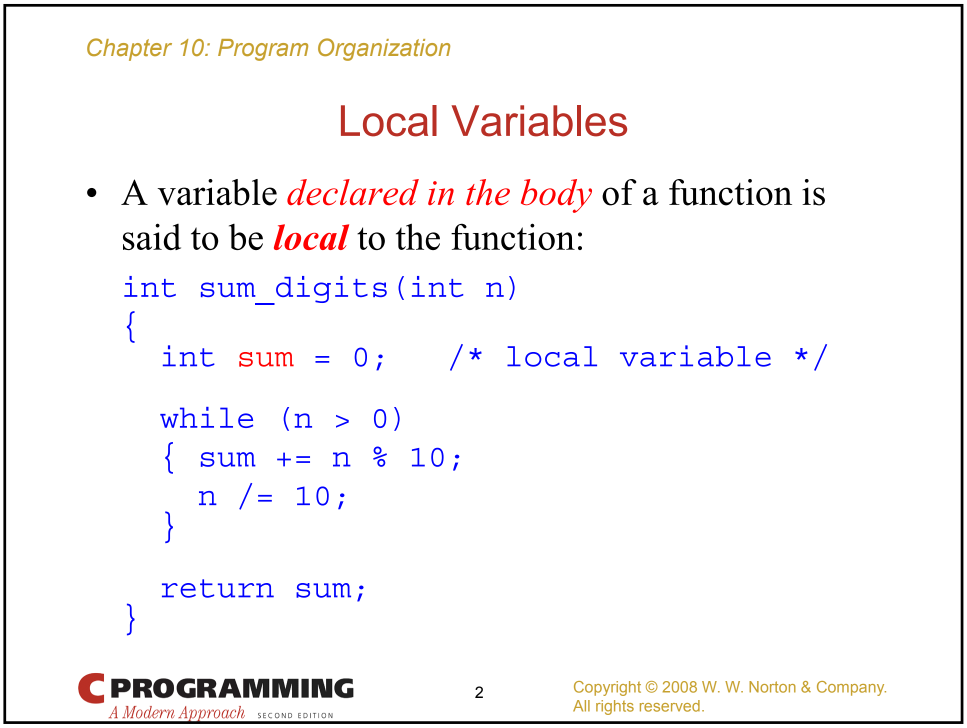


Chapter 10

Program Organization



Local Variables

- A variable *declared in the body* of a function is said to be ***local*** to the function:

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

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

  return sum;
}
```

Local Variables

- Default properties of local variables:
  - ***Automatic storage duration***

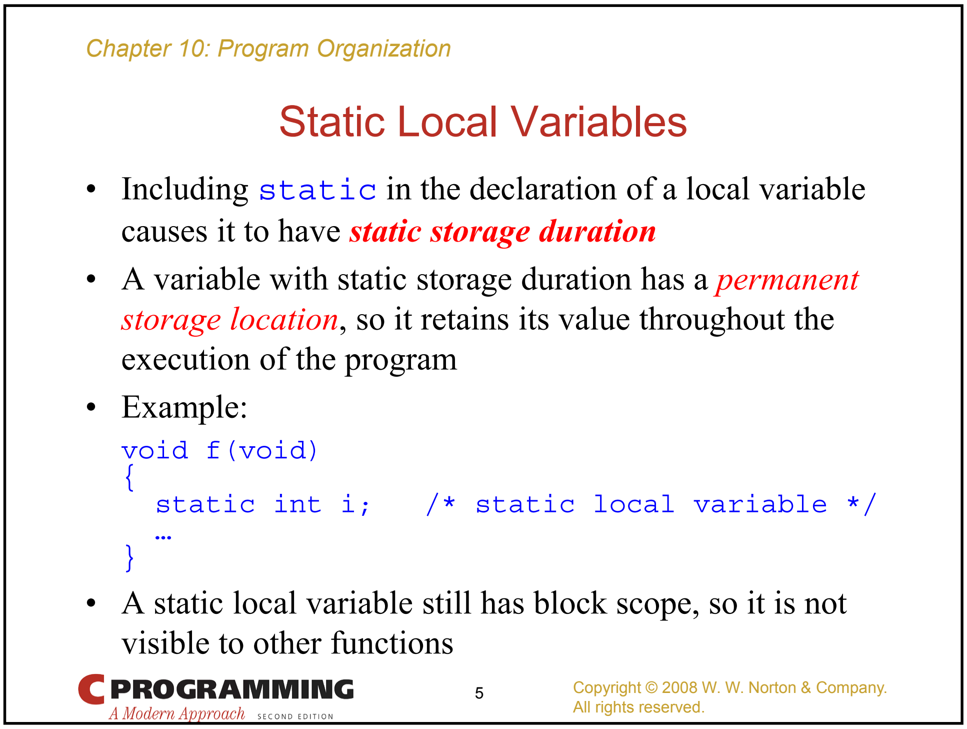
    Storage is "*automatically*"
    - ***allocated*** when the enclosing function is called and
    - ***deallocated*** when the function returns
  - ***Block scope***

    A local variable is ***visible***
    - from its point of declaration
    - to the end of the enclosing function body

Local Variables

- Since C99 does not require variable declarations to come at the beginning of a function, it is possible for a local variable to have a very small scope:

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

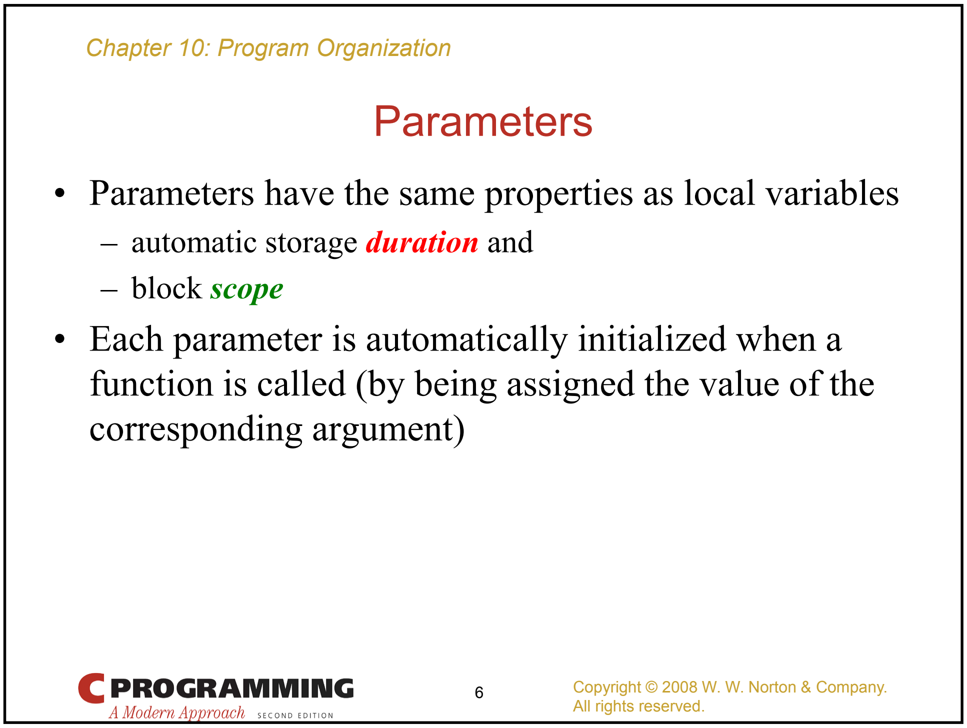


Static Local Variables

- Including `static` in the declaration of a local variable causes it to have ***static storage duration***
- A variable with static storage duration has a *permanent storage location*, so it retains its value throughout the execution of the program
- Example:

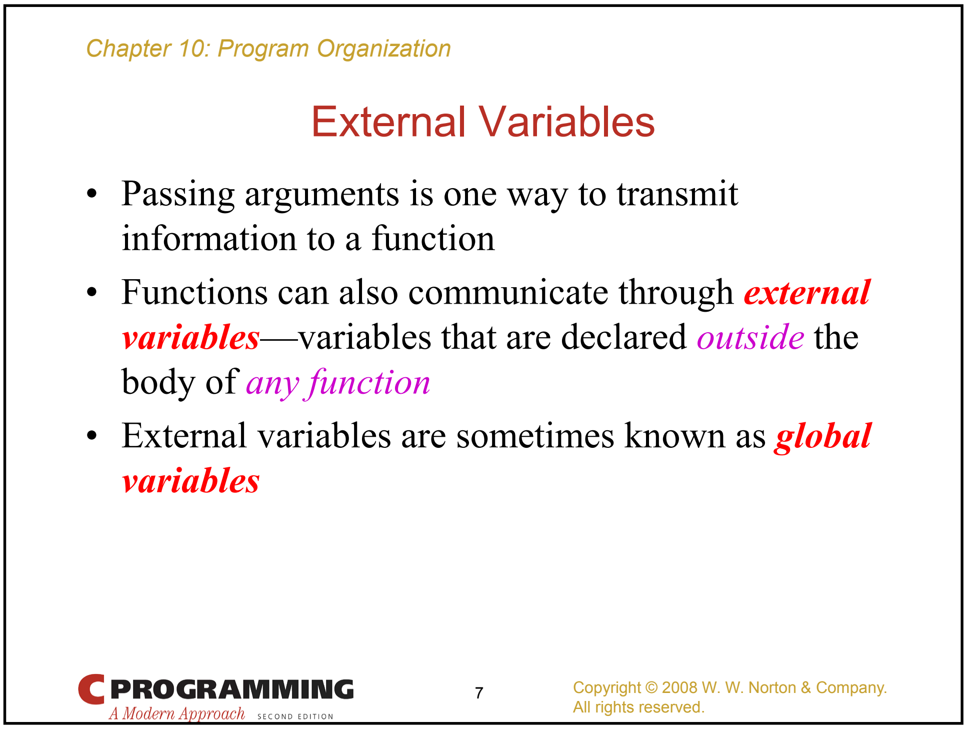
```
void f(void)
{
  static int i;    /* static local variable */
  …
}
```

- A static local variable still has block scope, so it is not visible to other functions



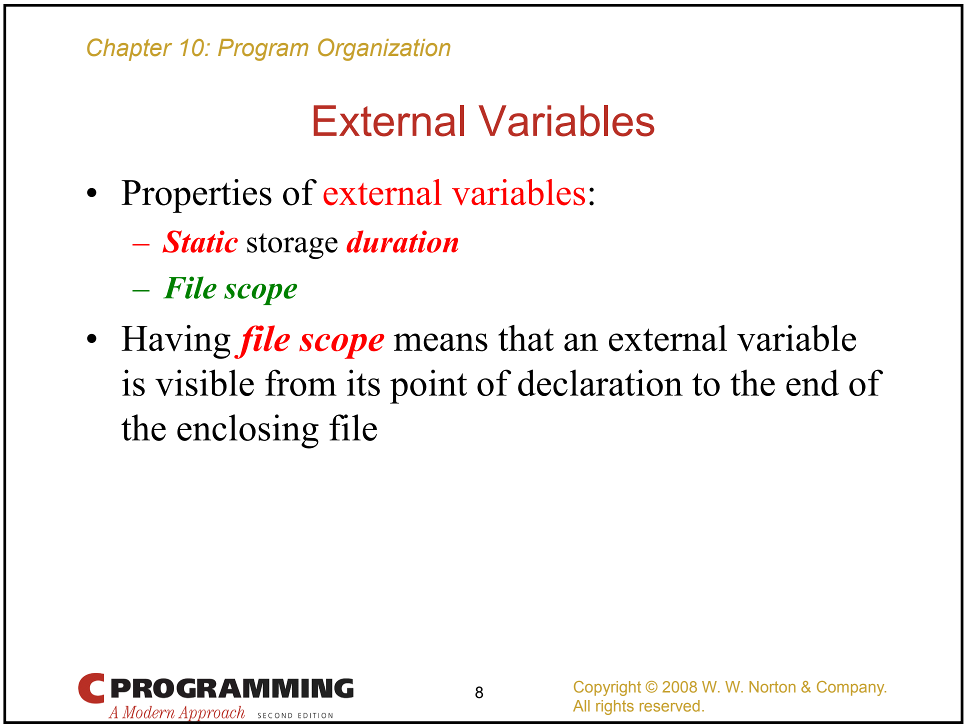
Parameters

- Parameters have the same properties as local variables
  - automatic storage ***duration*** and
  - block ***scope***
- Each parameter is automatically initialized when a function is called (by being assigned the value of the corresponding argument)



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*
- External variables are sometimes known as ***global variables***



External Variables

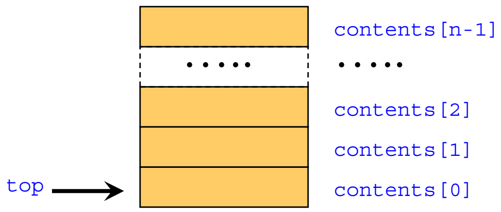
- Properties of external variables:
  - ***Static*** storage ***duration***
  - ***File scope***
- Having ***file scope*** means that an external variable is visible from its point of declaration to the end of the enclosing file

Example: Using External Variables to Implement a Stack

- To illustrate how external variables might be used, let us look at a data structure known as a ***stack***
- A stack, like an array, can store multiple data items of the same type
- The operations on a stack are limited:
  - ***Push*** an item (add it to one end—the "stack top")
  - ***Pop*** an item (remove it from the same end)
- Examining or modifying an item that is *not* at the top of the stack is forbidden

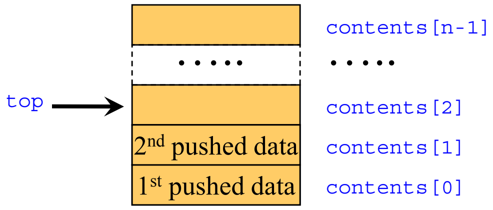
Example: Using External Variables to Implement a Stack

- One way to implement a stack in `C` is to store its items in an array, which we will 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



Example: Using External Variables to Implement a Stack

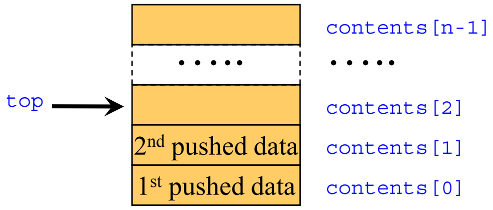
- To *push* an item:
  - Store it in `contents` at the position indicated by `top`, then
  - increment `top`



contents[n-1]
• • • • •
top
contents[2]
2nd pushed data
contents[1]
1st pushed data
contents[0]

Example: Using External Variables to Implement a Stack

- To *pop* an item:
  - Decrement `top`
  - then use it as an index into `contents` to fetch the item that is being popped



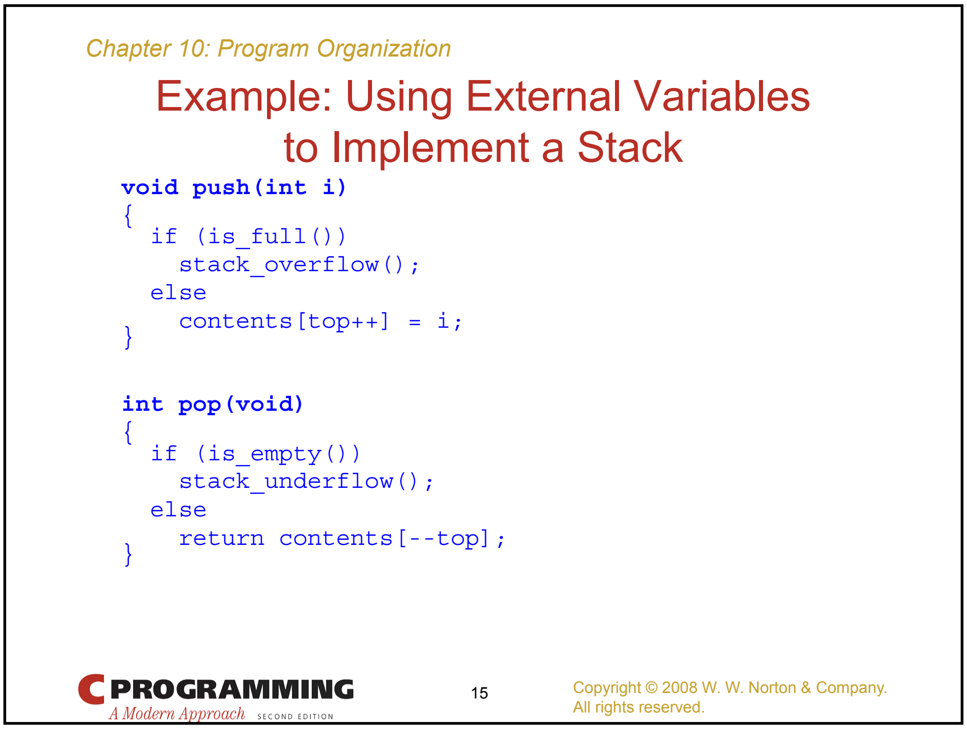
contents[n-1]
• • • • •
top
contents[2]
2nd pushed data
contents[1]
1st pushed data
contents[0]

Example: Using External Variables to Implement a Stack

- The following program fragment
  - declares the `contents` and `top` variables for a stack
  - provides a set of functions that represent stack operations
- All five functions need access to the `top` variable, and two functions need access to `contents`, so `contents` and `top` will be external

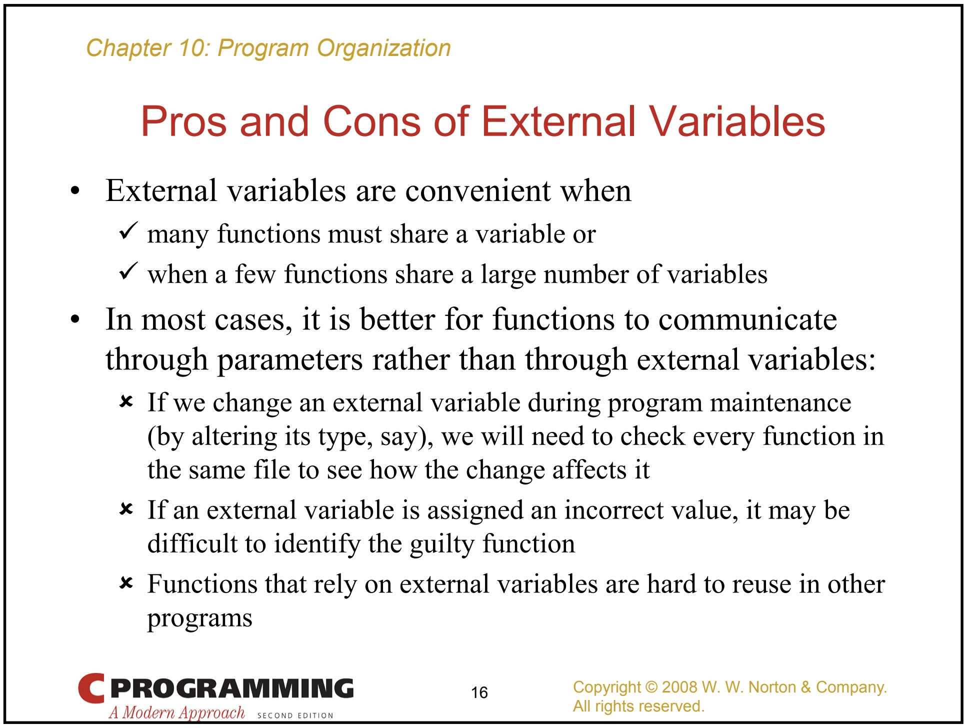
Example: Using External Variables to Implement a Stack

```
#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;
}
```



Example: Using External Variables to Implement a Stack

```
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, it is better for functions to communicate through parameters rather than through external variables:
  - ✗ If we change an external variable during program maintenance (by altering its type, say), we will 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
  - ✗ Functions that rely on external variables are hard to reuse in other programs

Pros and Cons of External Variables

- Making variables external when they should be local can lead to some rather frustrating bugs
- Code that 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

Pros and Cons of External Variables

- Do not use 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 just once at the top of the program
  - This practice is misleading; someone reading the program later may think that the uses of i are related, when in fact they are not

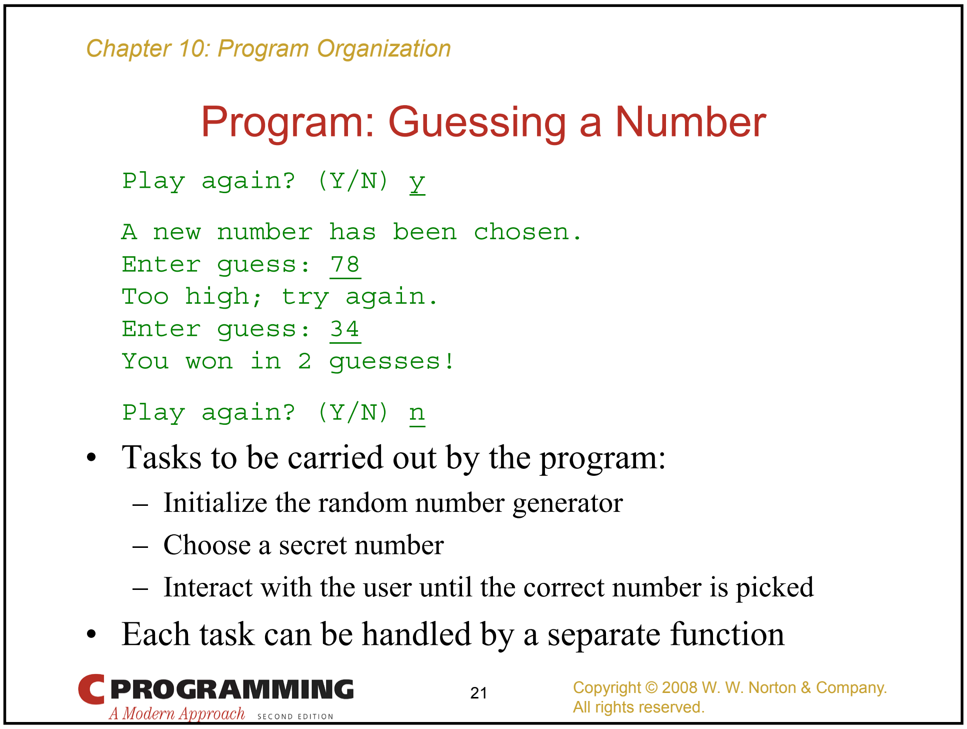
Pros and Cons of External Variables

- Make sure that external variables have meaningful names
- Local variables do not always need meaningful names: it is often hard to think of a better name than `i` for the control variable in a `for` loop

Program: Guessing a Number

- The `guess.c` program generates a random number between 1 and 100, which the user attempts to guess in as few tries as possible:

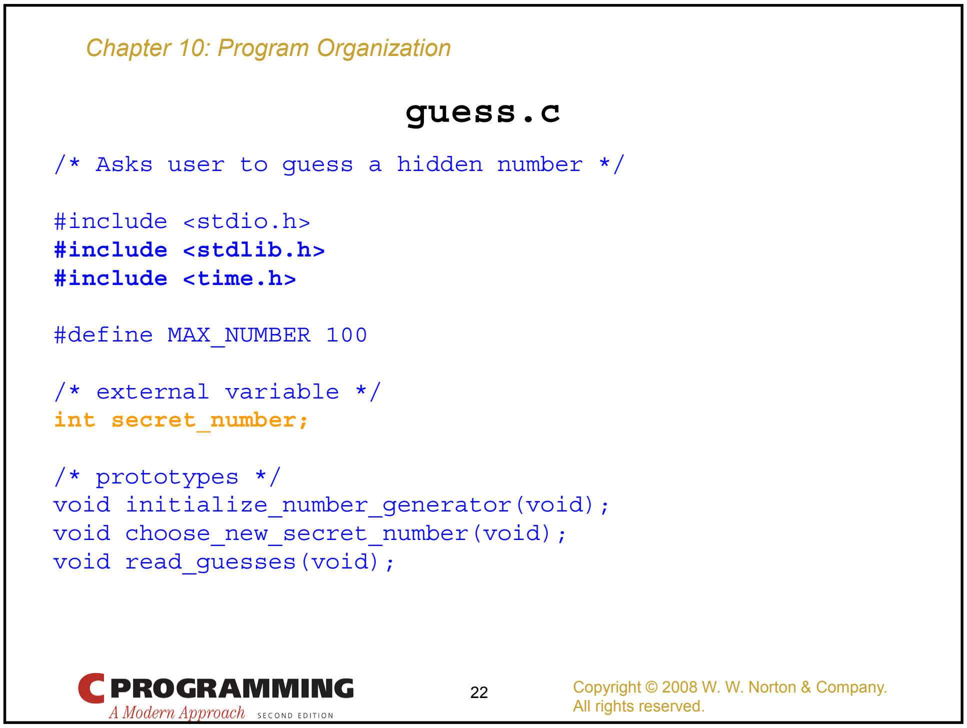
```
Guess the secret number between 1 and 100.

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!
```



Program: Guessing a Number

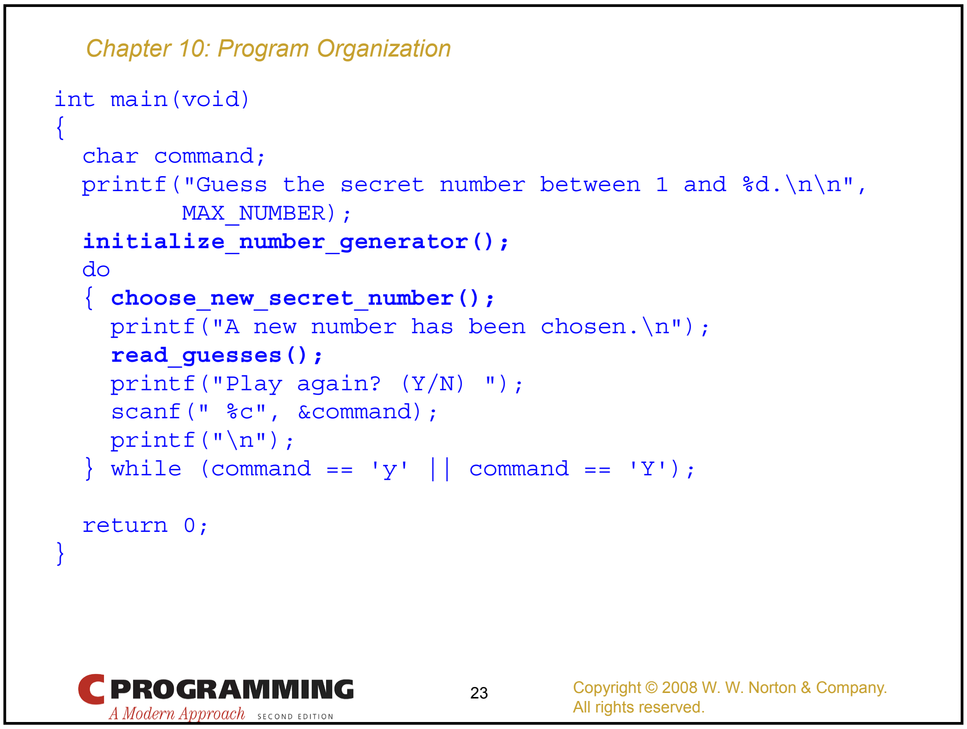
```
Play again? (Y/N) y

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
```

- Tasks to be carried out by the program:
  - Initialize the random number generator
  - Choose a secret number
  - Interact with the user until the correct number is picked
- Each task can be handled by a separate function

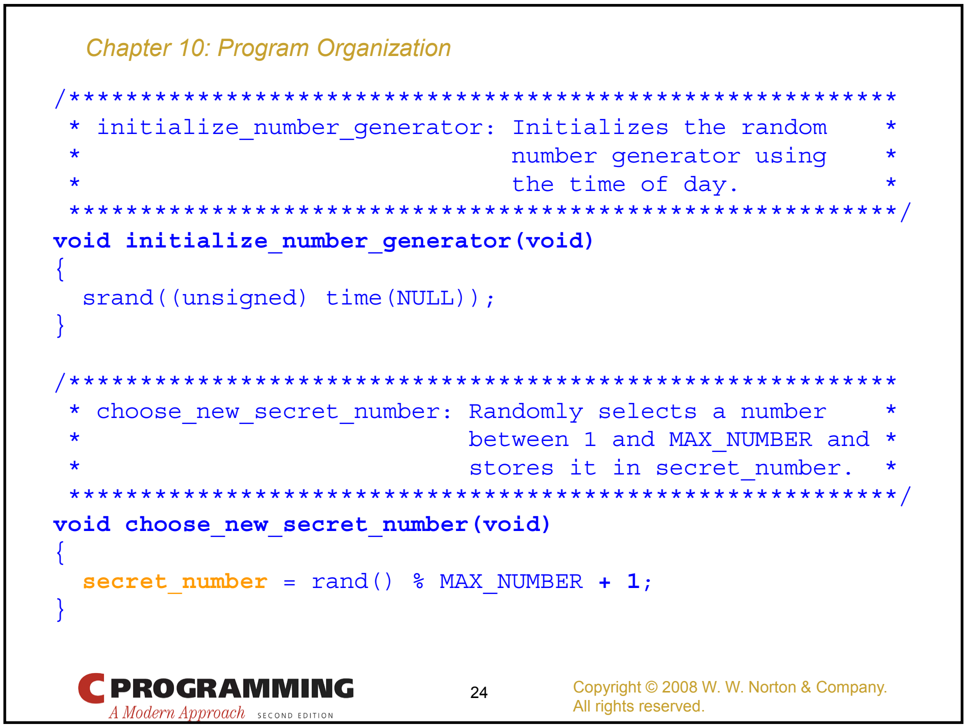


guess.c

```
/* 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;
}
```

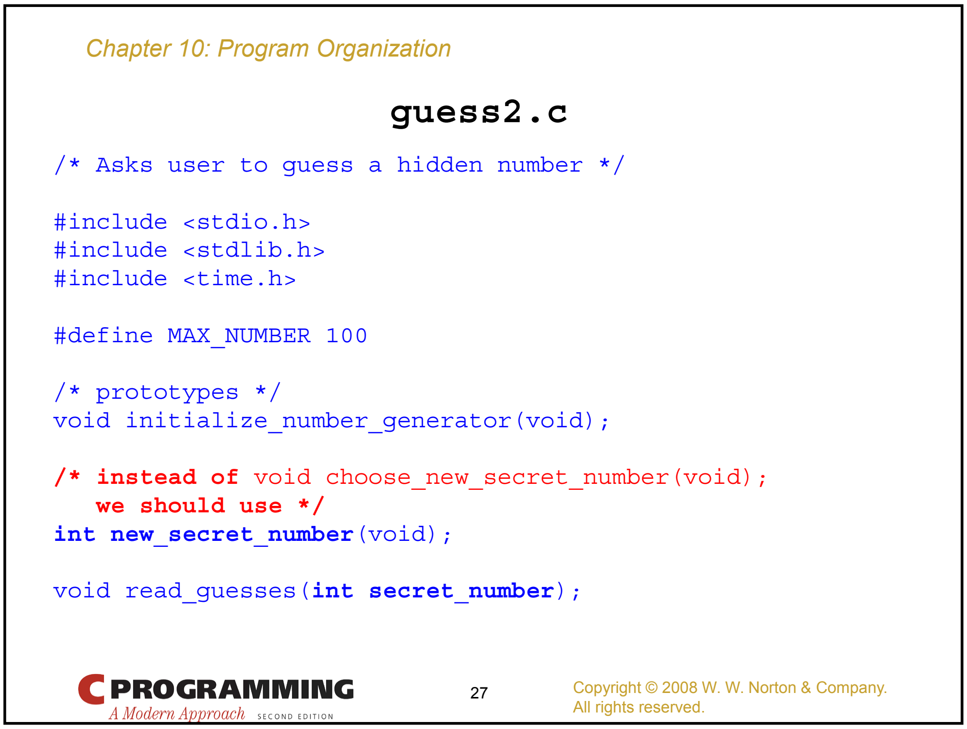


```
/**********************************************************
 * initialize_number_generator: Initializes the random    *
 *                              number generator using    *
 *                              the time of day.          *
 **********************************************************/
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");
  }
}
```

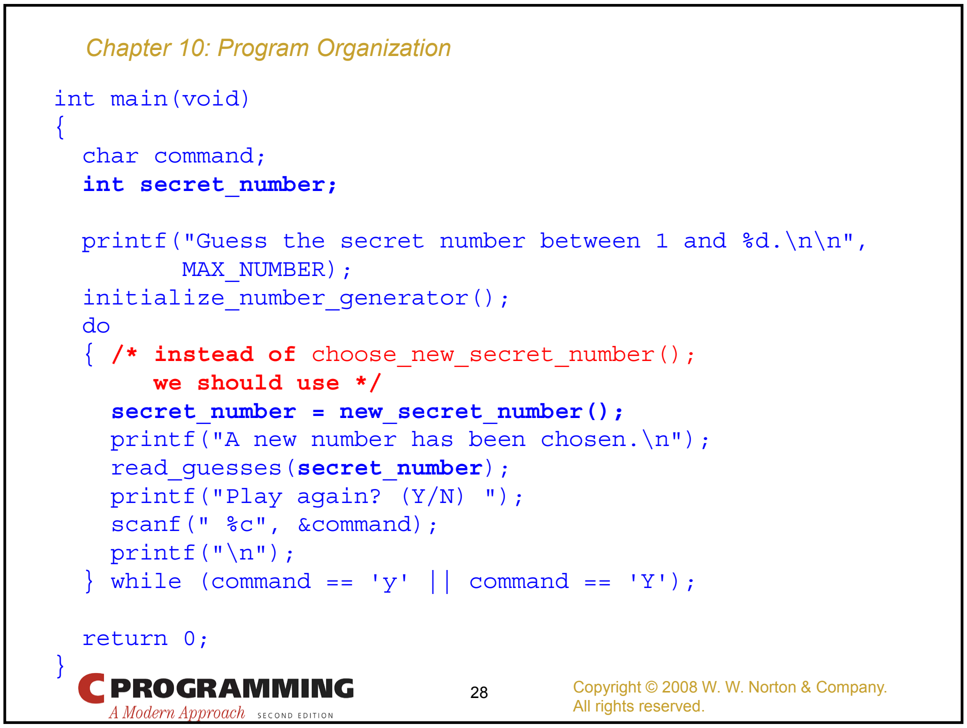
Program: Guessing a Number

- Although `guess.c` works fine, it relies on the external variable `secret_number`
- By altering `choose_new_secret_number` and `read_guesses` slightly, we can move `secret_number` into the `main` function
- The new version of `guess.c` follows

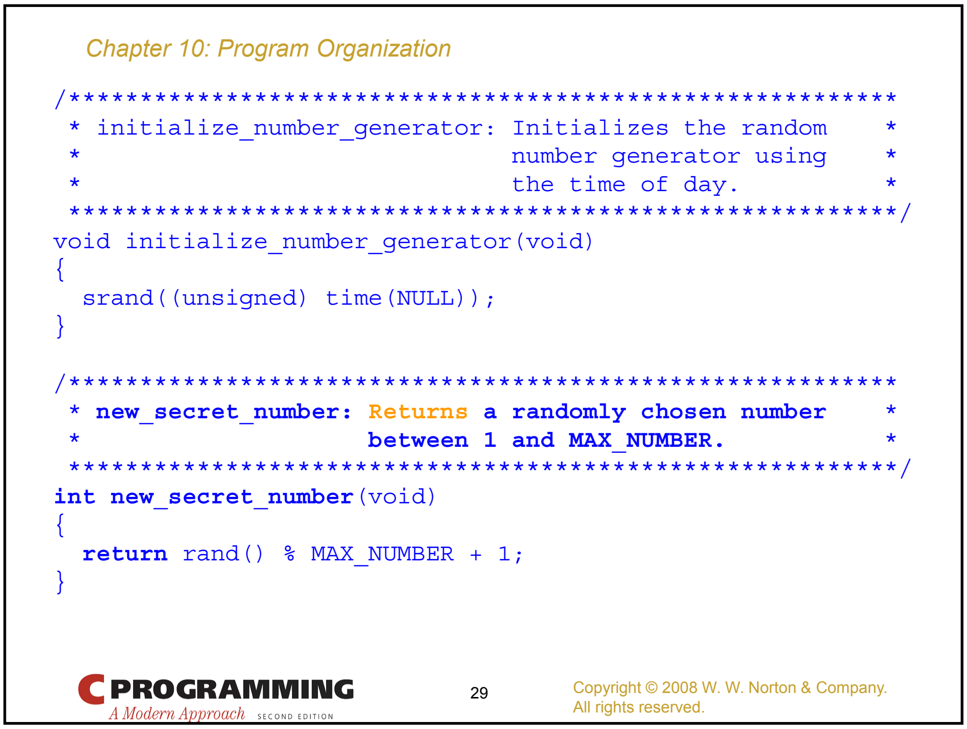


guess2.c

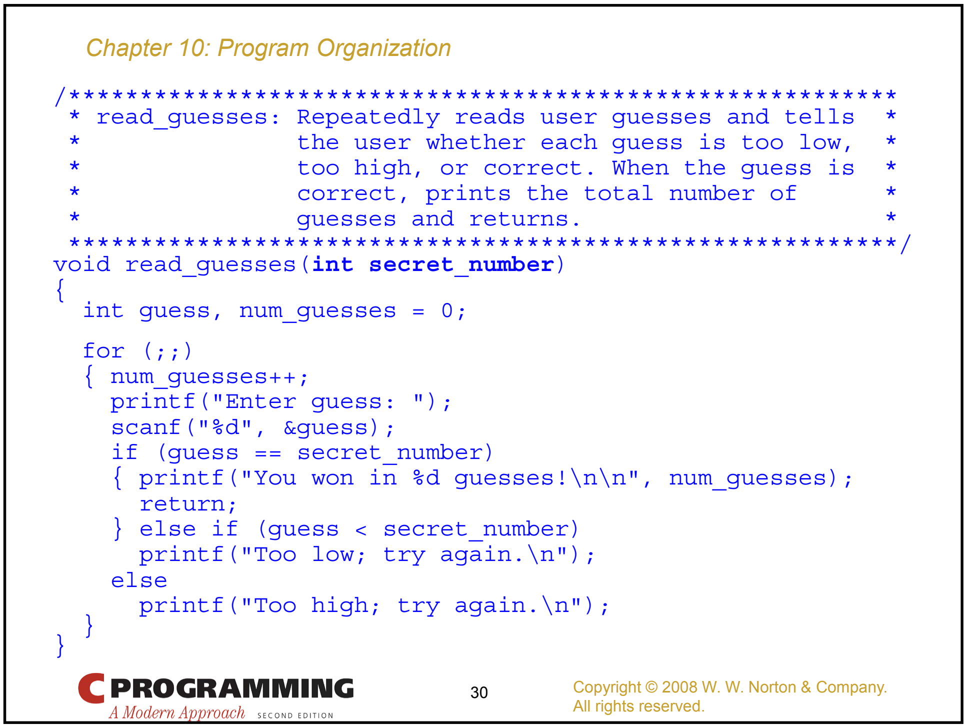
```
/* Asks user to guess a hidden number */

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

#define MAX_NUMBER 100

/* prototypes */
void initialize_number_generator(void);

/* instead of void choose_new_secret_number(void);
   we should use */
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
  { /* instead of choose_new_secret_number();
       we should use */
    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");
  }
}
```

Blocks

- In Section 5.2, we encountered compound statements of the form

```
{
  statements
}
```

- **C** allows compound statements to contain *declarations* as well as *statements*:

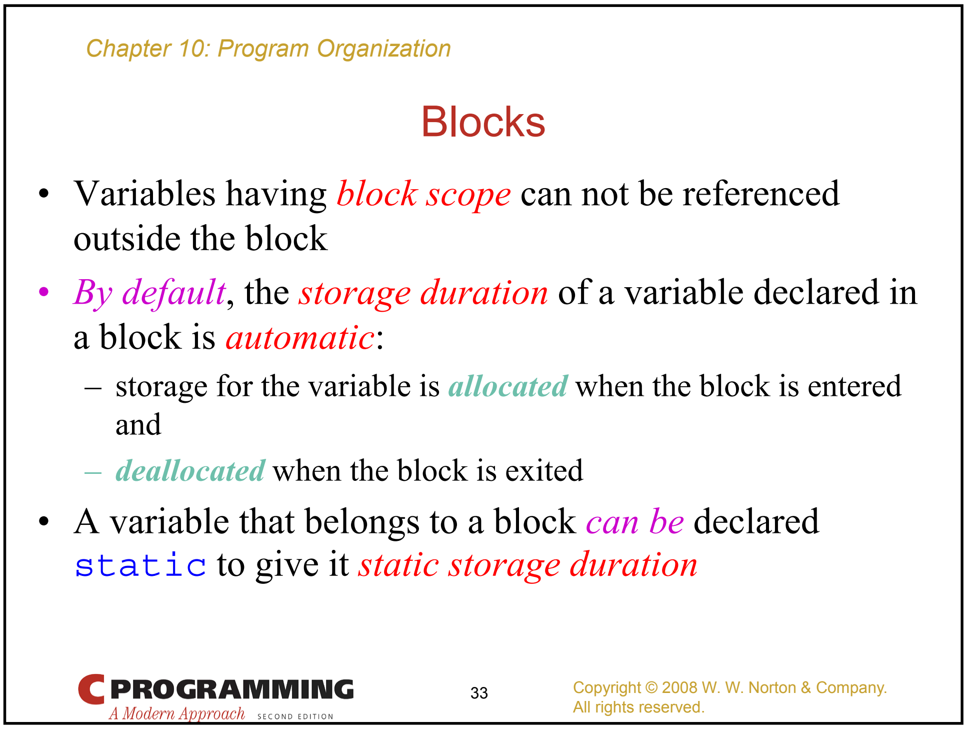
```
{
  declarations
  statements
}
```

- This kind of compound statement is called a ***block***

Blocks

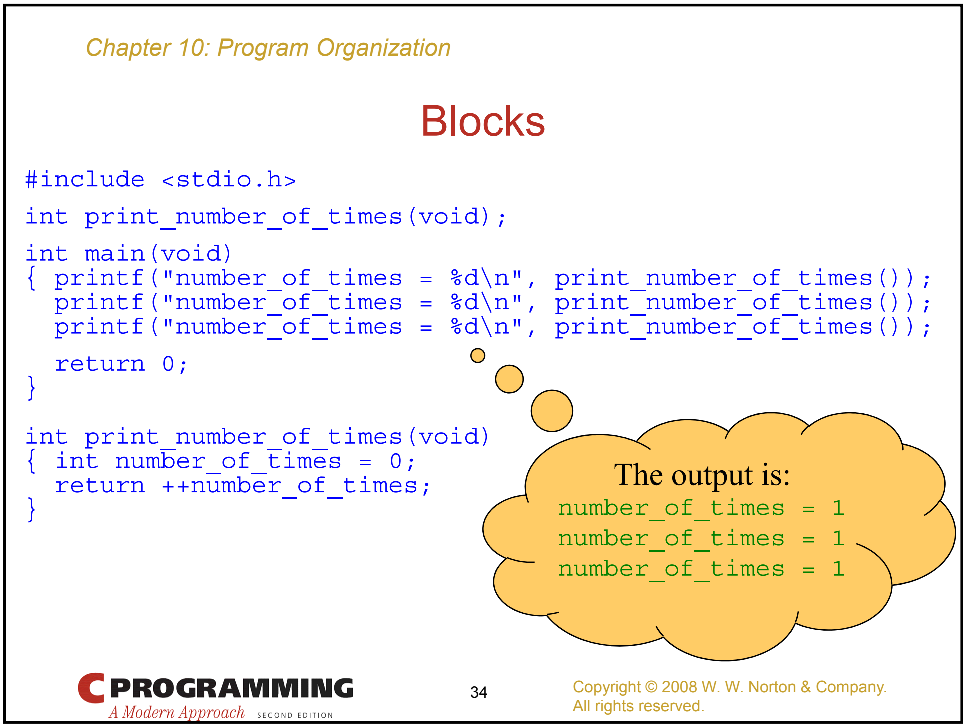
- Example of a block:

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

  i = j;
  j = temp;
}
```



Blocks

- Variables having *block scope* can not be referenced outside the block
- *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
- A variable that belongs to a block *can be* declared `static` to give it *static storage duration*

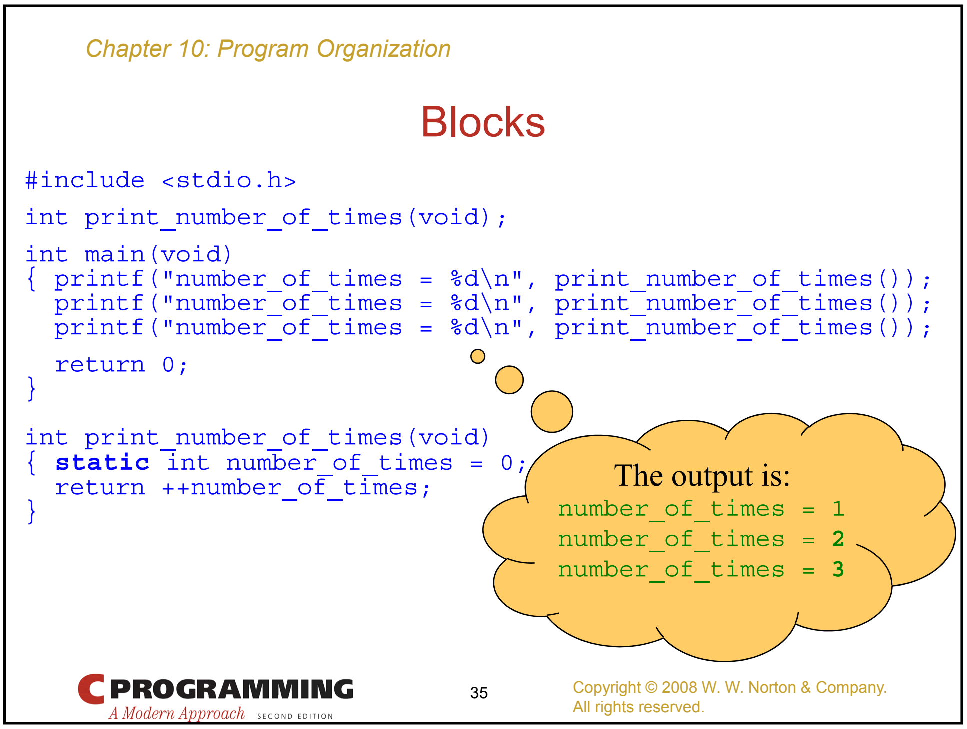


Blocks

```
#include <stdio.h>
int print_number_of_times(void);
int main(void)
{ printf("number_of_times = %d\n", print_number_of_times());
  printf("number_of_times = %d\n", print_number_of_times());
  printf("number_of_times = %d\n", print_number_of_times());
  return 0;
}

int print_number_of_times(void)
{ int number_of_times = 0;
  return ++number_of_times;
}
```

The output is:

```
number_of_times = 1
number_of_times = 1
number_of_times = 1
```

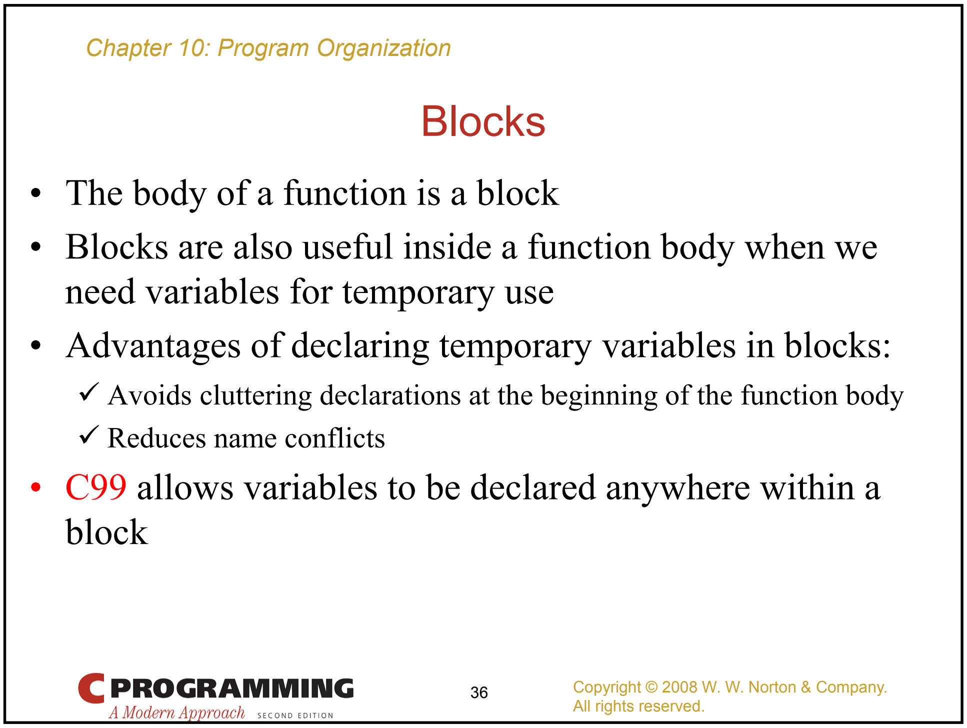


Blocks

```
#include <stdio.h>

int print_number_of_times(void);

int main(void)
{ printf("number_of_times = %d\n", print_number_of_times());
  printf("number_of_times = %d\n", print_number_of_times());
  printf("number_of_times = %d\n", print_number_of_times());

  return 0;
}

int print_number_of_times(void)
{ static int number_of_times = 0;
  return ++number_of_times;
}
```

The output is:

```
number_of_times = 1
number_of_times = 2
number_of_times = 3
```

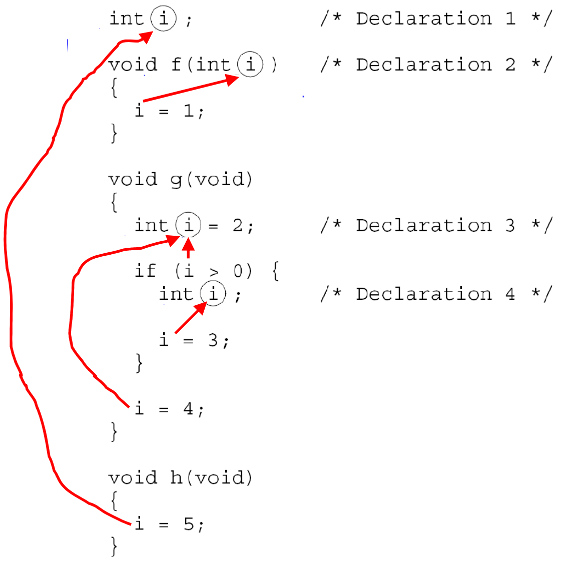


Blocks

- The body of a function is a block
- Blocks are also useful inside a function body when we need variables for temporary use
- Advantages of declaring temporary variables in blocks:
  - ✓ Avoids cluttering declarations at the beginning of the function body
  - ✓ Reduces name conflicts
- C99 allows variables to be declared anywhere within a block

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
- *The most important scope rule*:
  - When a declaration inside a block names an identifier that is *already visible*,
    - 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



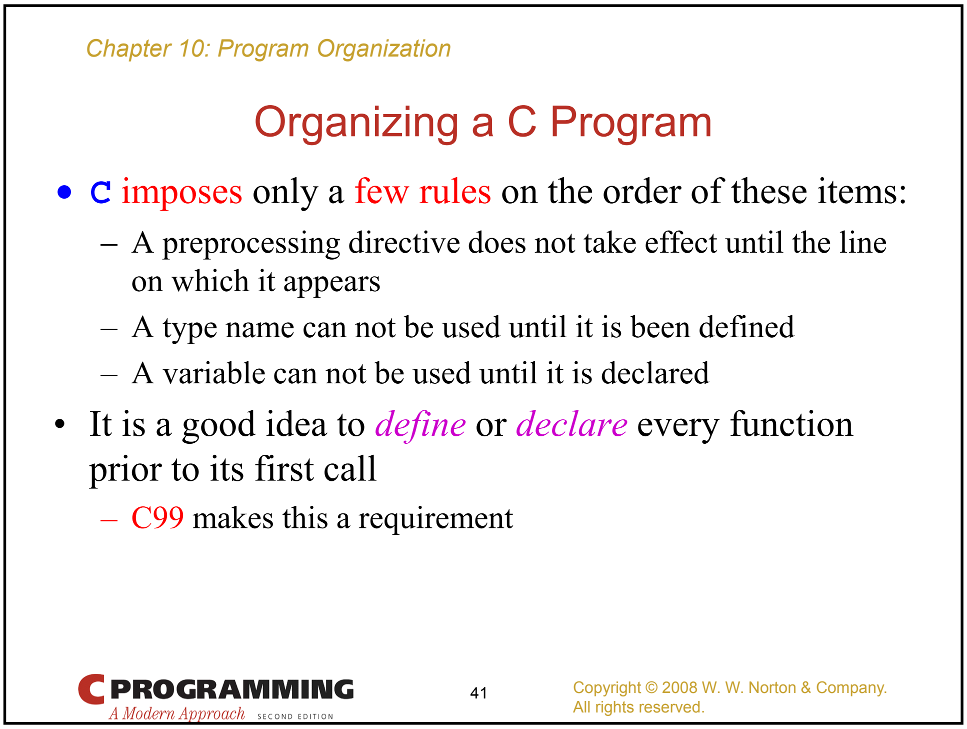
```
int i;             /* Declaration 1 */

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;
}
```

Scope

- In the example on the previous slide, the identifier `i` has four different meanings:
  - In Declaration 1, `i` is a variable with static storage duration and file scope
  - 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
- `C`’s scope rules allow us to determine the meaning of `i` each time it is used

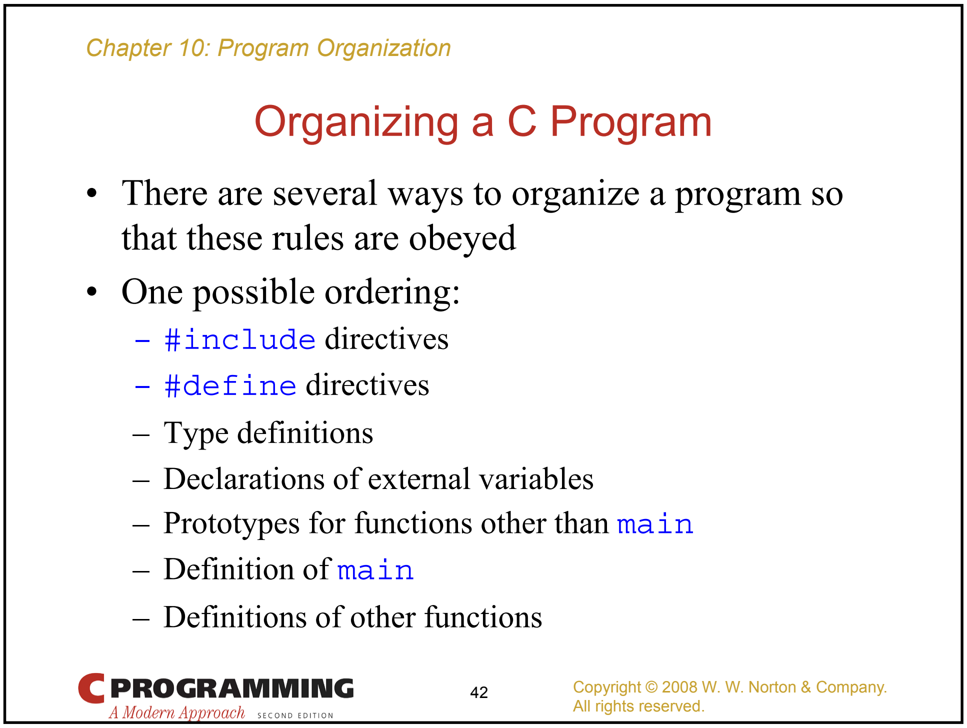
Organizing a C Program

- Major elements of a `C` program:
  - Preprocessing directives such as `#include` and `#define`
  - Type definitions
  - Declarations of external variables
  - Function prototypes
  - Function definitions



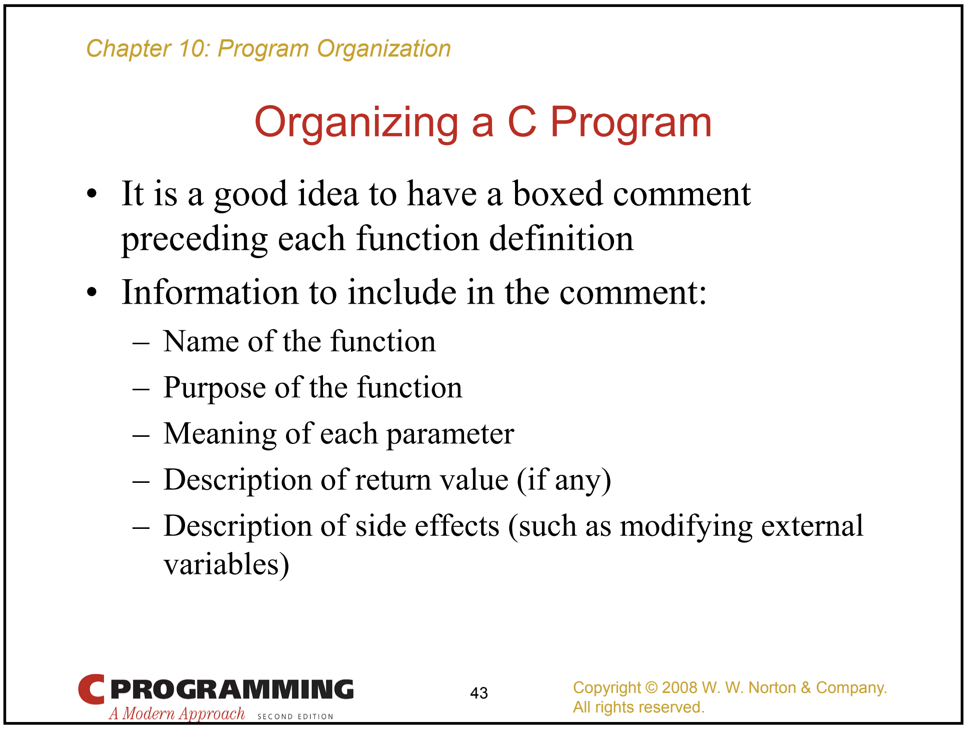
Organizing a C Program

- `C` imposes only a few rules on the order of these items:
  - A preprocessing directive does not take effect until the line on which it appears
  - A type name can not be used until it is been defined
  - A variable can not be used until it is declared
- It is a good idea to *define* or *declare* every function prior to its first call
  - C99 makes this a requirement



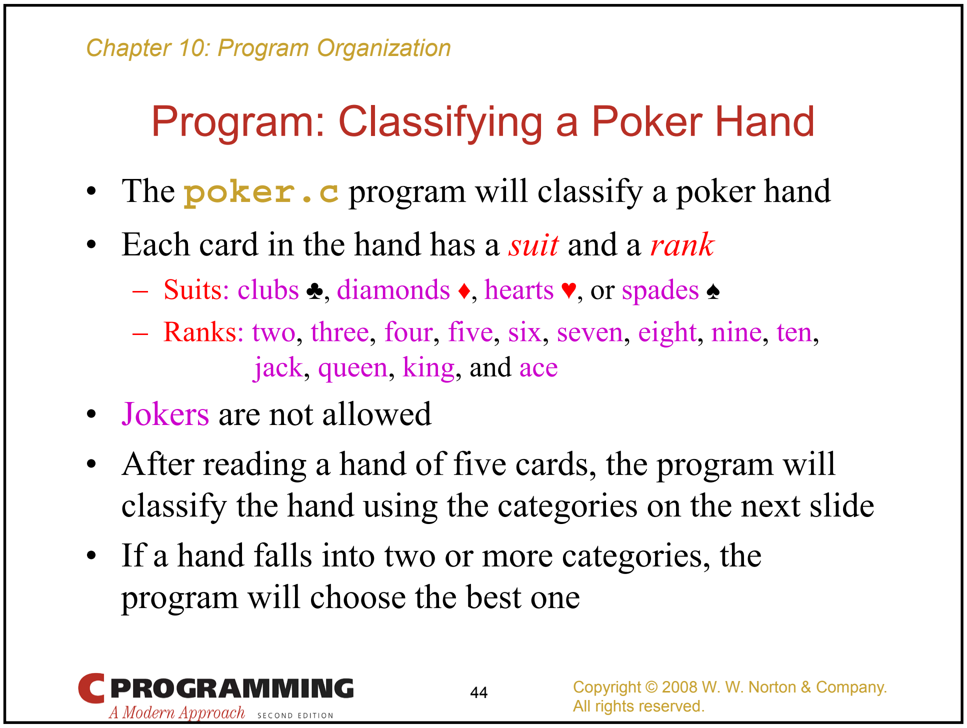
Organizing a C Program

- There are several ways to organize a program so that these rules are obeyed
- 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



Organizing a C Program

- It is a good idea to have a boxed comment preceding each function definition
- Information to include in the comment:
  - Name of the function
  - Purpose of the function
  - Meaning of each parameter
  - Description of return value (if any)
  - Description of side effects (such as modifying external variables)



Program: Classifying a Poker Hand

- The `poker.c` program will classify a poker hand
- Each card in the hand has a *suit* and a *rank*
  - Suits: clubs ♣, diamonds ♦, hearts ♥, or spades ♠
  - Ranks: two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, and ace
- Jokers are not allowed
- After reading a hand of five cards, the program will classify the hand using the categories on the next slide
- If a hand falls into two or more categories, the program will choose the best one

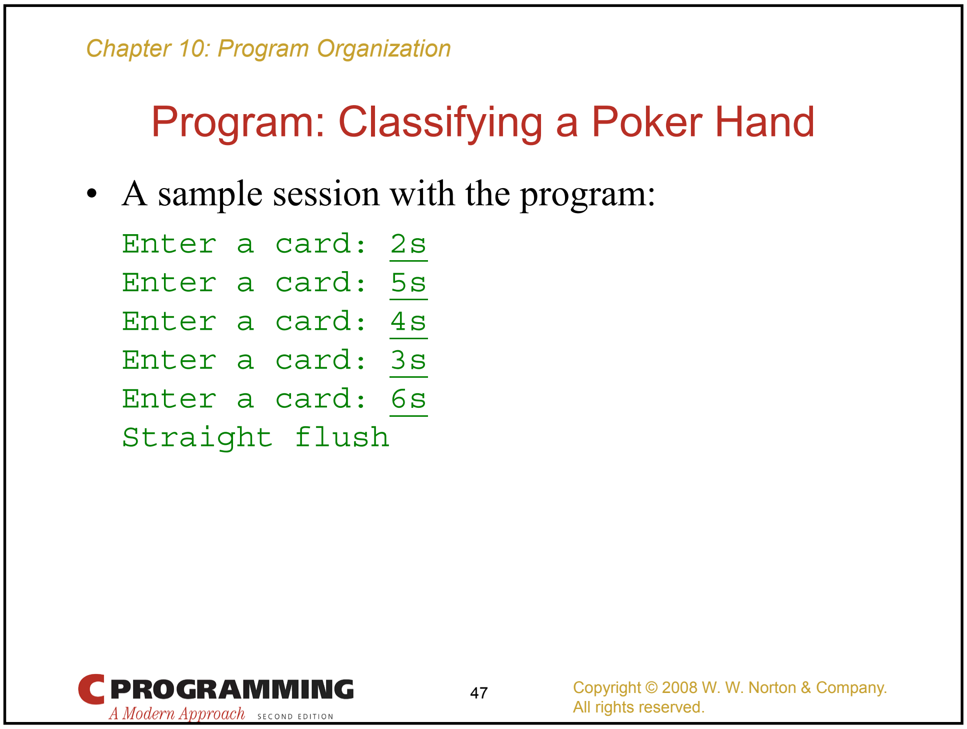
Program: Classifying a Poker Hand

- Categories (listed 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)

Program: Classifying a Poker Hand

- For input purposes, ranks and suits will be single letters (upper- or lower-case):

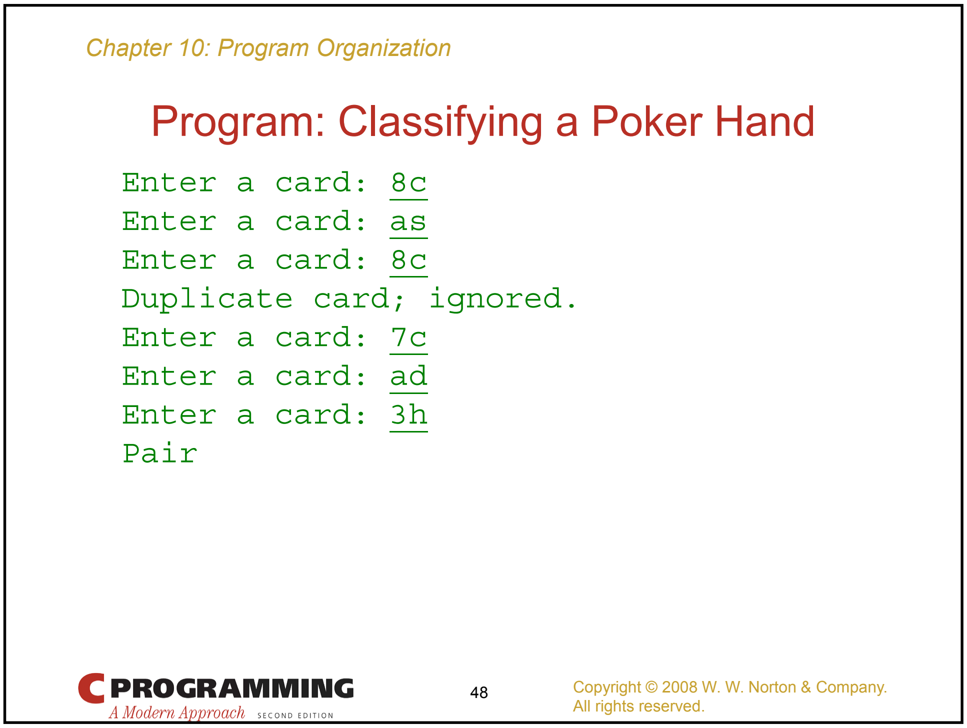
  Ranks: `2 3 4 5 6 7 8 9 t j q k a`
  Suits: `c d h s`
- Actions to be taken if the user enters an illegal card or tries to enter the same card twice:
  - Ignore the card
  - Issue an error message
  - Request another card
- Entering the number 0 instead of a card will cause the program to terminate



Program: Classifying a Poker Hand

- A sample session with the program:

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



Program: Classifying a Poker Hand

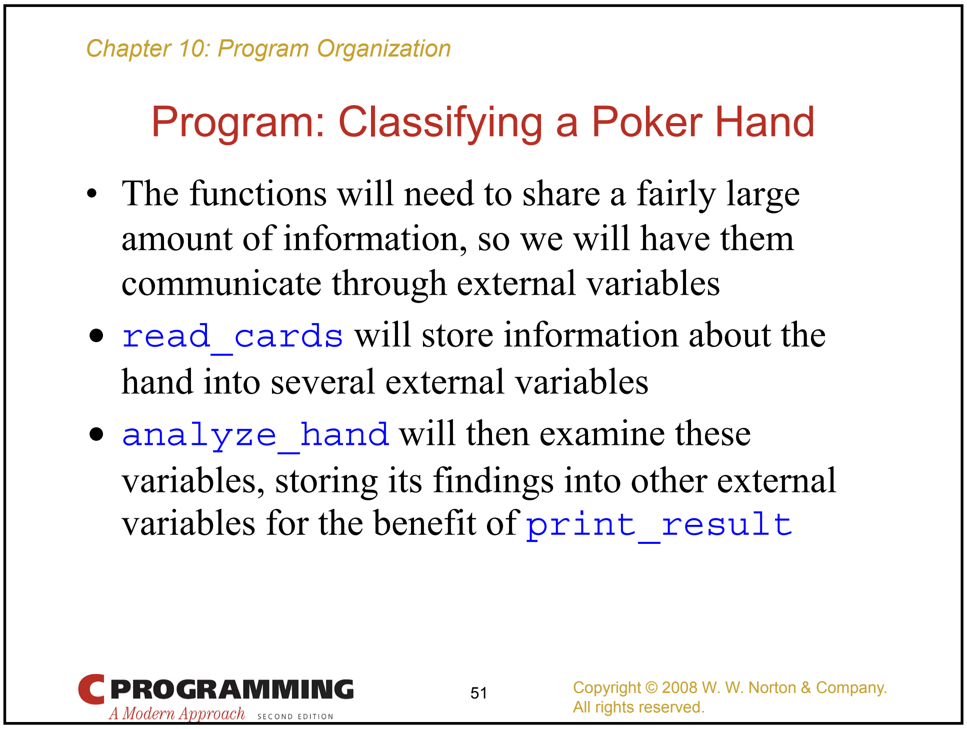
```
Enter a card: 8c
Enter a card: as
Enter a card: 8c
Duplicate card; ignored.
Enter a card: 7c
Enter a card: ad
Enter a card: 3h
Pair
```

Program: Classifying a Poker Hand

```
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
```

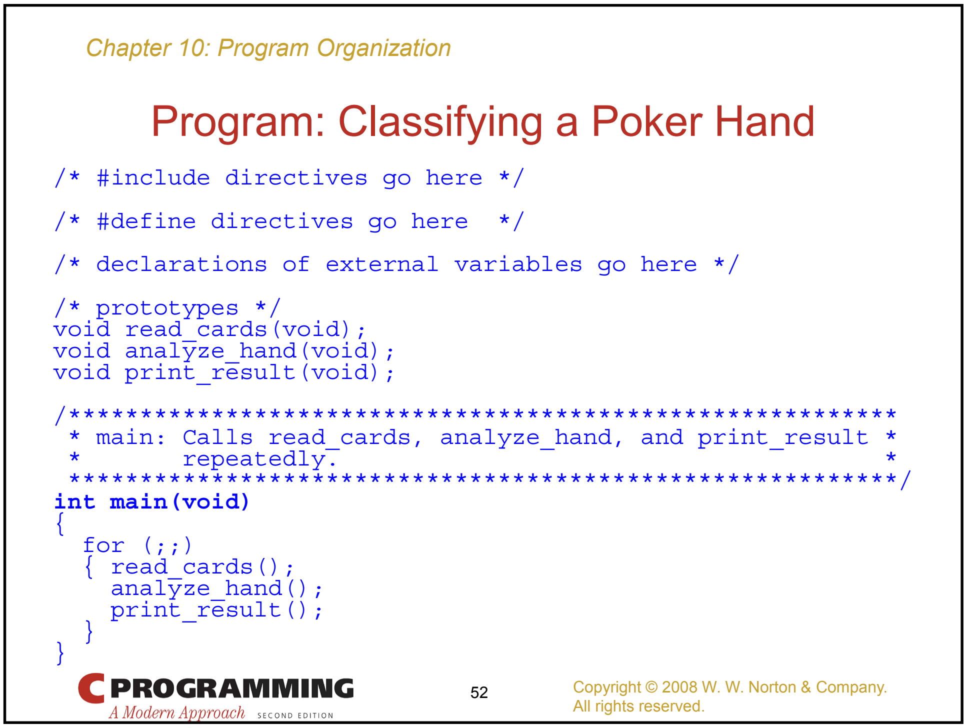
Program: Classifying a Poker Hand

- The program has three tasks:
  - Read a hand of five cards
  - Analyze the hand for pairs, straights, and so forth
  - Print the classification of the hand
- The functions `read_cards`, `analyze_hand`, and `print_result` will perform these tasks
- The `main` function does nothing but call these functions inside an endless loop



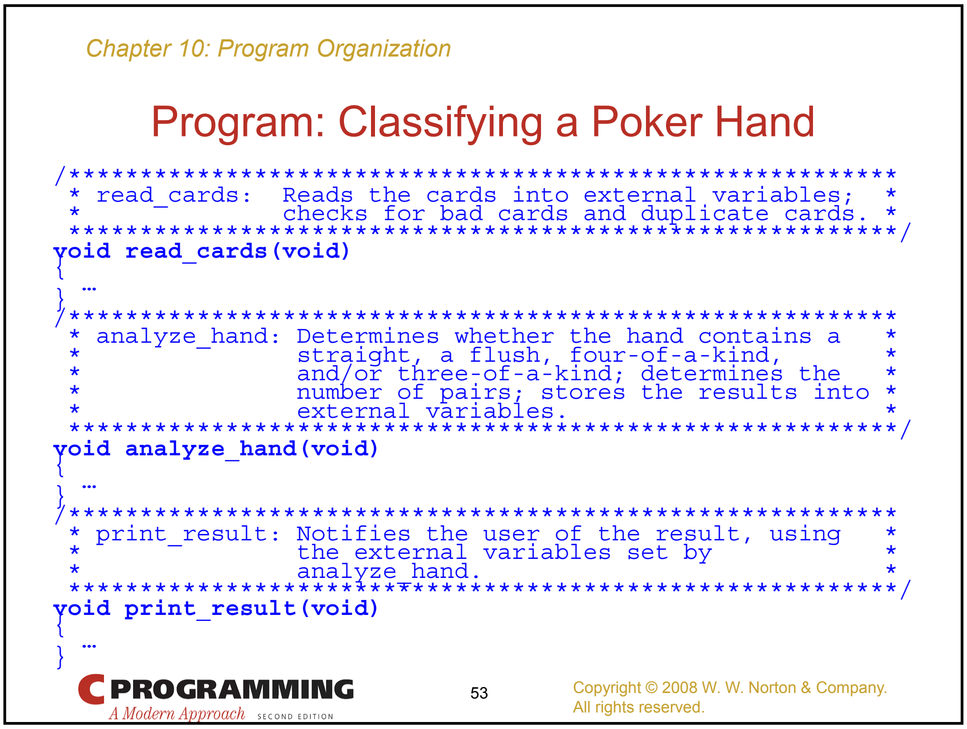
Program: Classifying a Poker Hand

- The functions will need to share a fairly large amount of information, so we will 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`



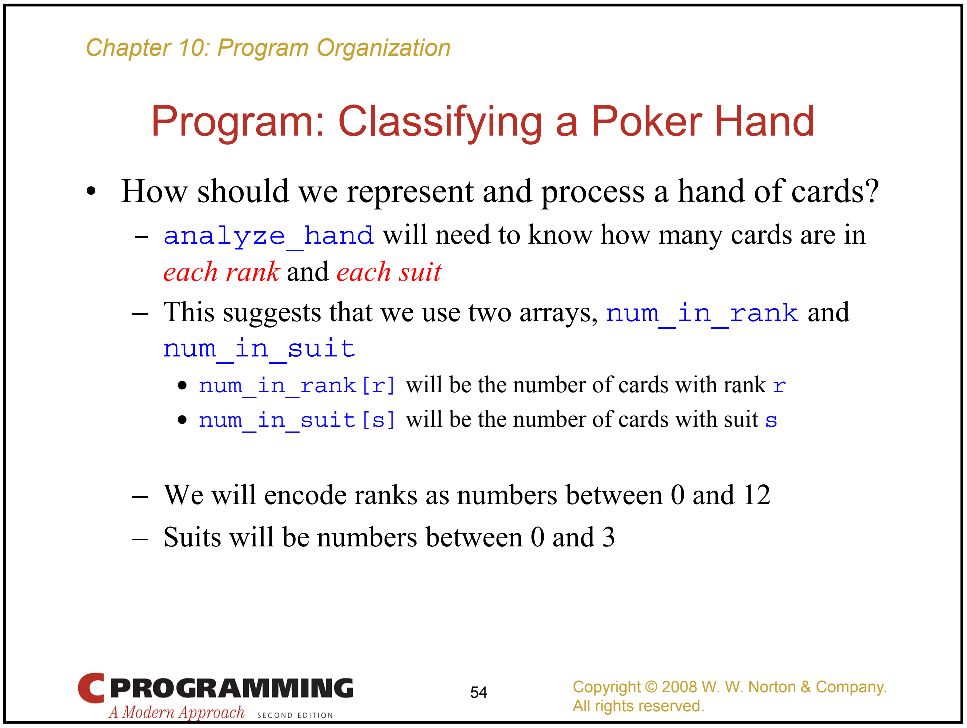
Program: Classifying a Poker Hand

```
/* #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();
  }
}
```



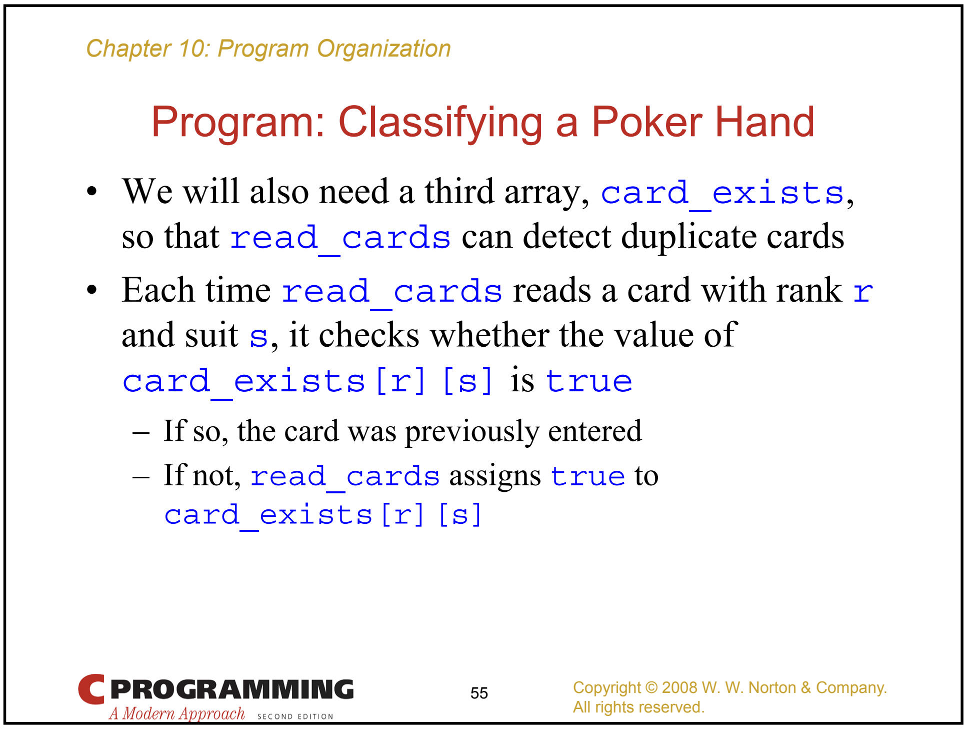
Program: Classifying a Poker Hand

```
/**********************************************************
 * 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)
{
 …
}
```



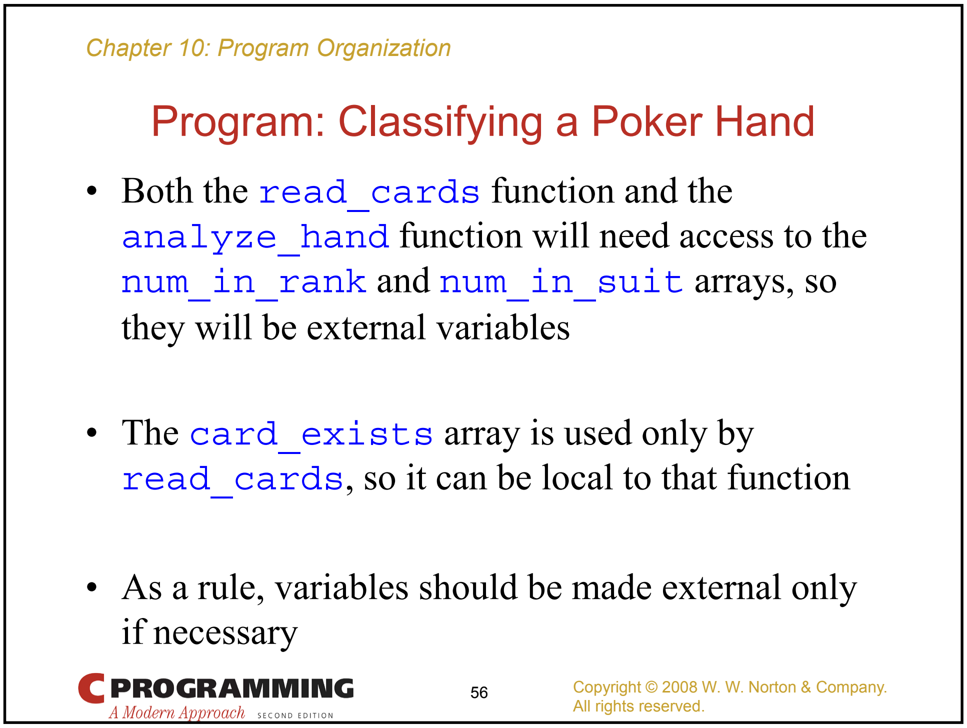
Program: Classifying a Poker Hand

- How should we represent and process a hand of cards?
  - `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`
    - `num_in_rank[r]` will be the number of cards with rank `r`
    - `num_in_suit[s]` will be the number of cards with suit `s`
  - We will encode ranks as numbers between 0 and 12
  - Suits will be numbers between 0 and 3



Program: Classifying a Poker Hand

- We will 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]`



Program: Classifying a Poker Hand

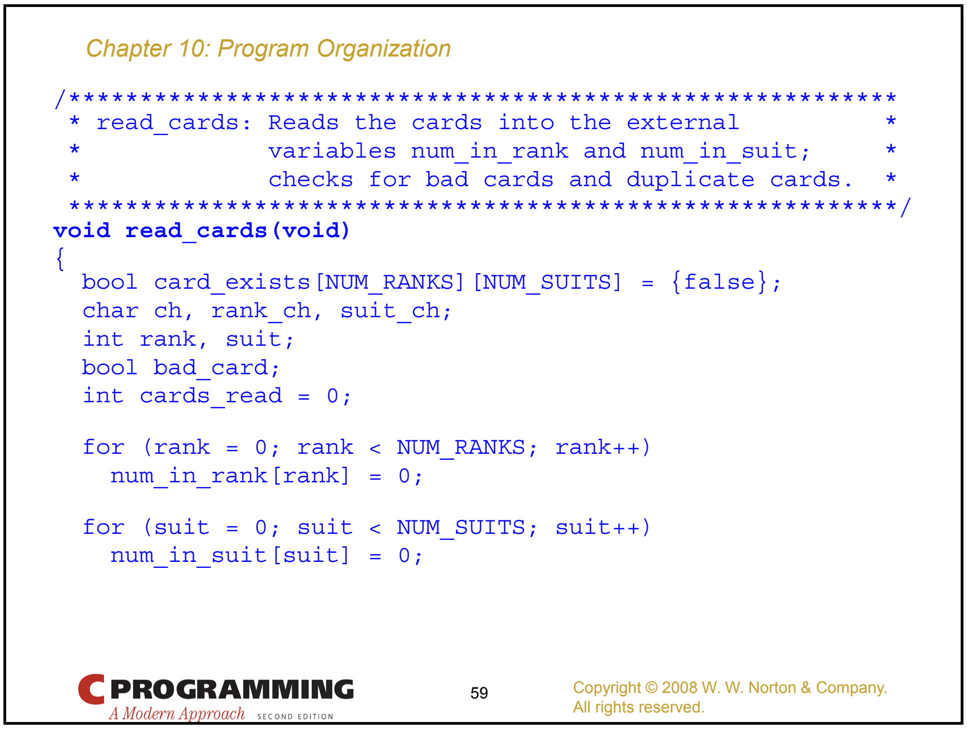
- Both the `read_cards` function and the `analyze_hand` function will need access to the `num_in_rank` and `num_in_suit` arrays, so they will be external variables
- The `card_exists` array is used only by `read_cards`, so it can be local to that function
- As a rule, variables should be made external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

poker.c

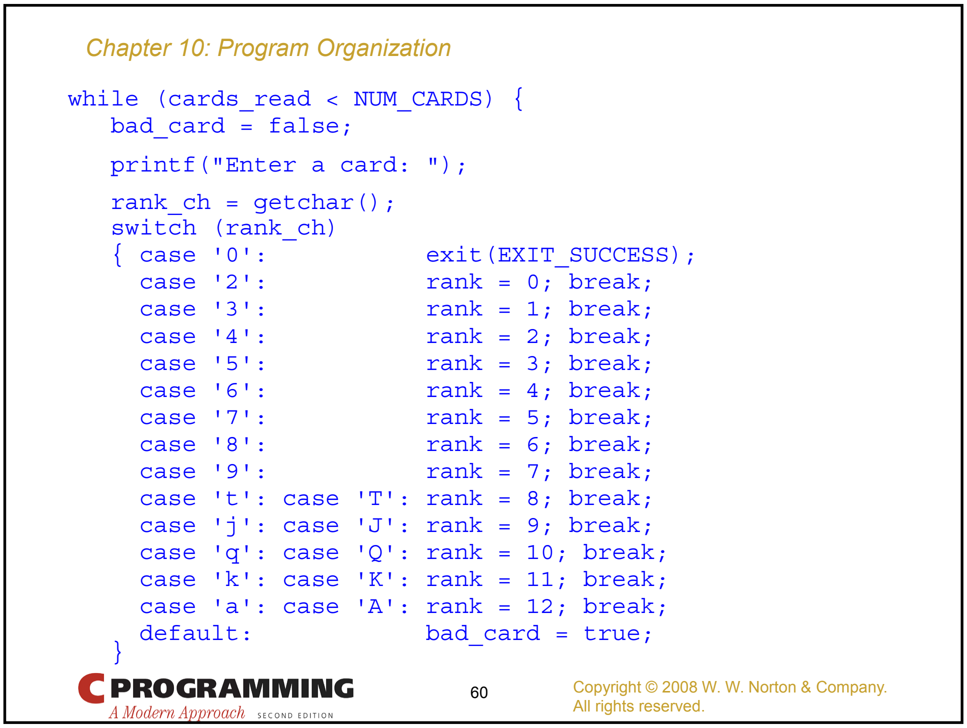
```
/* Classifies a poker hand */

#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 */
```

Why did not we initialize these variables?

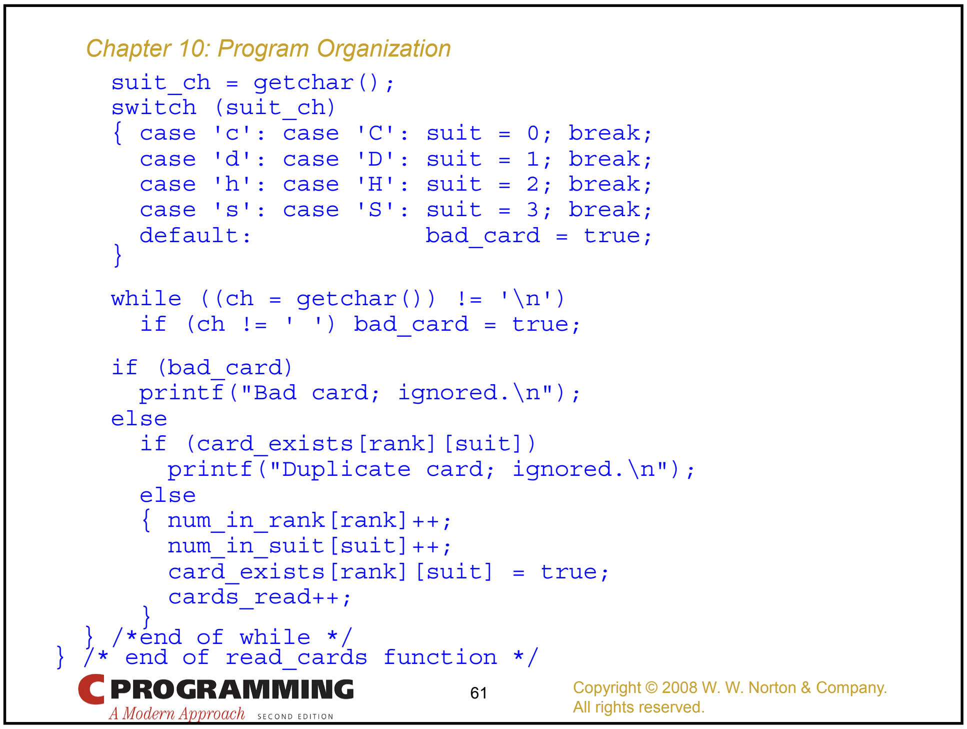
```
/* prototypes */
void read_cards(void);
void analyze_hand(void);
void print_result(void);

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



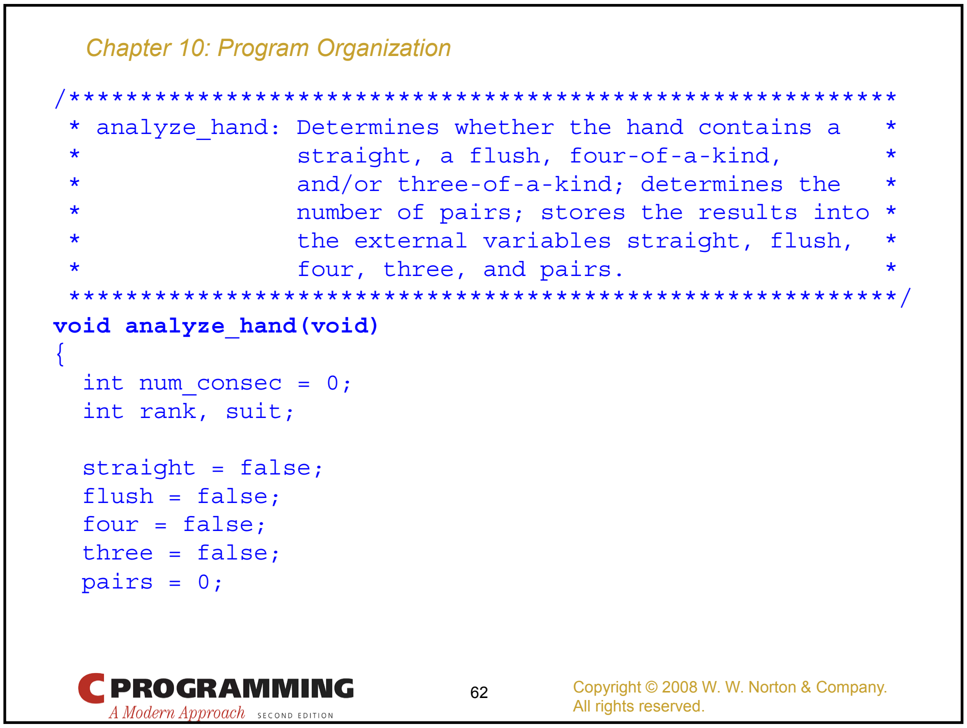
```
/**********************************************************
 * read_cards: Reads the cards into the external          *
 *             variables num_in_rank and num_in_suit;     *
 *             checks for bad cards and duplicate cards.  *
 **********************************************************/
void read_cards(void)
{
  bool card_exists[NUM_RANKS][NUM_SUITS] = {false};
  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++)
    num_in_suit[suit] = 0;
```



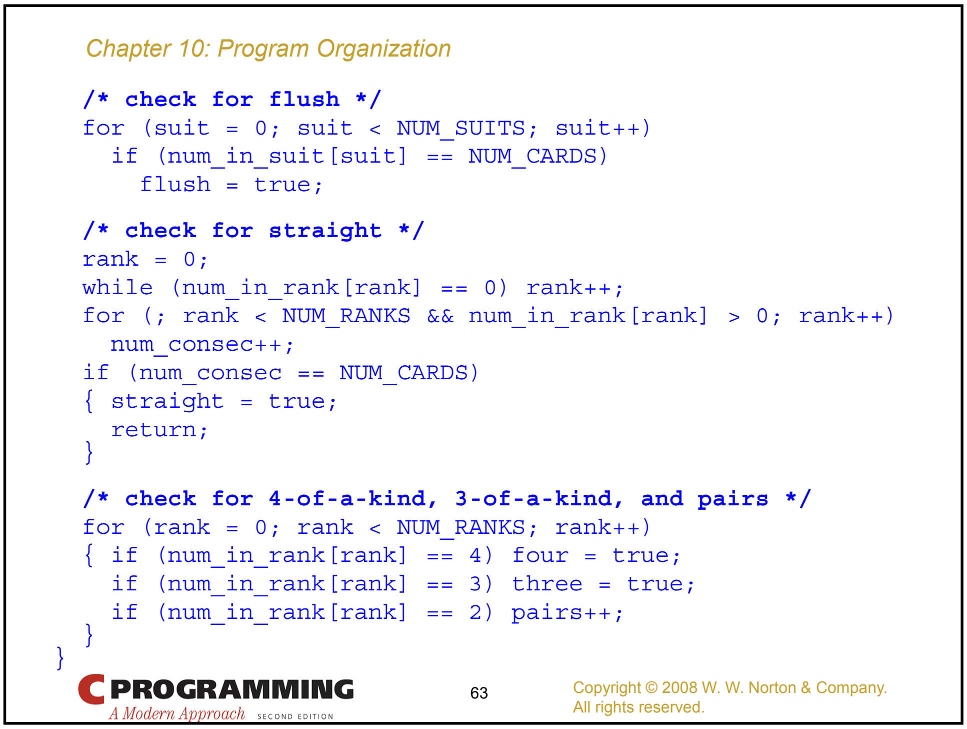
```
  while (cards_read < NUM_CARDS) {
    bad_card = false;

    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;
    }
```



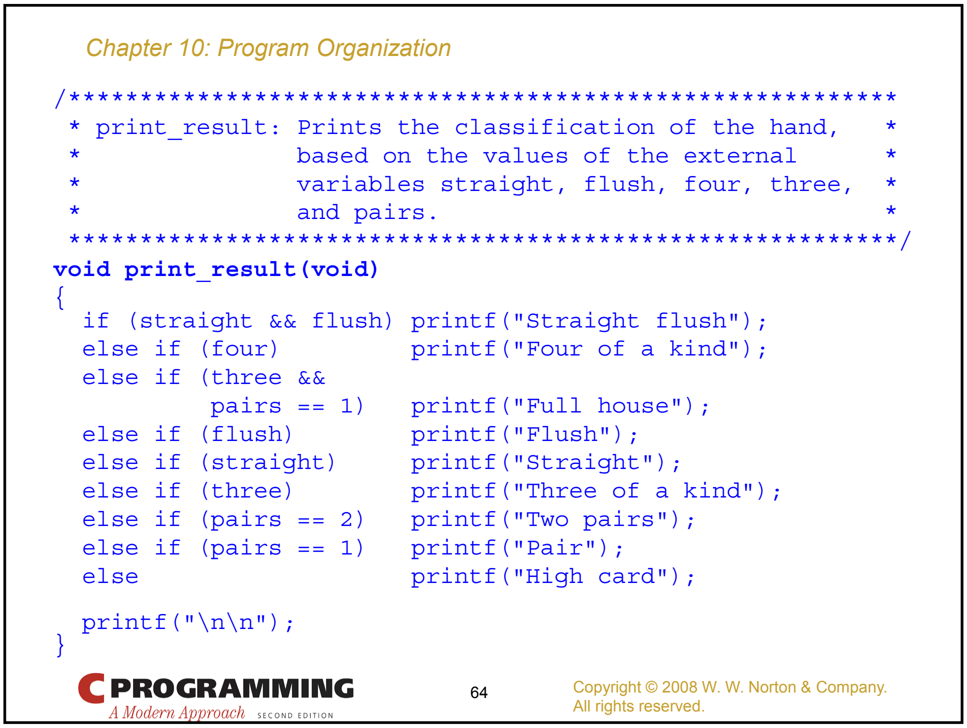
```
    suit_ch = getchar();
    switch (suit_ch)
    { case 'c': case 'C': suit = 0; break;
      case 'd': case 'D': suit = 1; break;
      case 'h': case 'H': suit = 2; break;
      case 's': case 'S': suit = 3; break;
      default:            bad_card = true;
    }

    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++;
      }
  } /*end of while */
} /* end of read_cards function */
```



```
/**********************************************************
 * 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");
}
```