Programming Practice (2)

2006. 04. 18
Introduction

- Pseudo code.
- Flowchart.
- Review & Programming Practice.
- Structured Programming
Pseudocode

- Artificial, informal language that helps us develop algorithms
- Similar to everyday English
- Not actually executed on computers
- Helps us “think out” a program before writing it
  - Easy to convert into a corresponding C++ program
  - Consists only of executable statements
Control Structures

- **Flowchart**
  - Graphical representation of an algorithm
  - Drawn using certain special-purpose symbols connected by arrows called flowlines
  - Rectangle symbol (action symbol):
    - Indicates any type of action
  - Oval symbol:
    - Indicates the beginning or end of a program or a section of code

- **Single-entry/single-exit control structures**
  - Connect exit point of one control structure to entry point of the next (control-structure stacking)
  - Makes programs easy to build
Selection structure:
- Used to choose among alternative courses of action
- Pseudocode:
  
  ```plaintext
  If student's grade is greater than or equal to 60
  Print "Passed"
  ```

If condition true
- Print statement executed and program goes on to next statement
- If false, print statement is ignored and the program goes onto the next statement
- Indenting makes programs easier to read
  - C ignores whitespace characters
The if Selection Statement

- **Pseudocode statement in C:**
  ```c
  if ( grade >= 60 )
      printf( "Passed\n" );
  ```
  - C code corresponds closely to the pseudocode
- **Diamond symbol (decision symbol)**
  - Indicates decision is to be made
  - Contains an expression that can be true or false
  - Test the condition, follow appropriate path
The if Selection Statement

- **if statement** is a single-entry/single-exit structure

- **if** statement is a single-entry/single-exit structure

- A decision can be made on any expression.
  - zero - false
  - nonzero - true

Example:

3 - 4 is true
The *if*...*else* Selection Statement

- **if**
  - Only performs an action if the condition is `true`

- **if...else**
  - Specifies an action to be performed both when the condition is `true` and when it is `false`

**Psuedocode:**

```plaintext
If student’s grade is greater than or equal to 60
    Print “Passed”
else
    Print “Failed”
```

- Note spacing/indentation conventions
The `if...else` Selection Statement

C code:
```c
if ( grade >= 60 )
    printf( "Passed\n" );
else
    printf( "Failed\n" );
```

Ternary conditional operator (`?:`)
- Takes three arguments (condition, value if true, value if false)
- Our pseudocode could be written:
  ```c
  printf( "%s\n", grade >= 60 ? "Passed" : "Failed" );
  ```
- Or it could have been written:
  ```c
  grade >= 60 ? printf( "Passed\n" ) : printf( "Failed\n" );
  ```
The `if...else` Selection Statement

- **Flow chart of the `if...else` selection statement**
  - `grade >= 60` (true) -> `print "Passed"`
  - `grade < 60` (false) -> `print "Failed"

- **Nested `if...else` statements**
  - Test for multiple cases by placing `if...else` selection statements inside `if...else` selection statement
  - Once condition is met, rest of statements skipped
  - Deep indentation usually not used in practice
The if...else Selection Statement

- **Pseudocode for a nested if...else statement**

  ```plaintext
  If student's grade is greater than or equal to 90
     Print “A”
  else
     If student's grade is greater than or equal to 80
        Print “B”
     else
        If student's grade is greater than or equal to 70
           Print “C”
        else
           If student's grade is greater than or equal to 60
              Print “D”
           else
              Print “F”
  ```
The if...else Selection Statement

- Compound statement:
  - Set of statements within a pair of braces
  - Example:
    ```c
    if ( grade >= 60 )
        printf( "Passed.\n" );
    else {
        printf( "Failed.\n" );
        printf( "You must take this course again.\n" );
    }
    ```

- Without the braces, the statement
  ```c
  printf( "You must take this course again.\n" );
  ```
  would be executed automatically
The if...else Selection Statement

- **Block:**
  - Compound statements with declarations

- **Syntax errors**
  - Caught by compiler

- **Logic errors:**
  - Have their effect at execution time
  - Non-fatal: program runs, but has incorrect output
  - Fatal: program exits prematurely
The **while** Repetition Statement

- **Repetition structure**
  - Programmer specifies an action to be repeated while some condition remains **true**
  - Psuedocode:
    
    ```
    While there are more items on my shopping list
    Purchase next item and cross it off my list
    ```
  
  - **while** loop repeated until condition becomes **false**
Example:

```java
int product = 2;
while (product <= 1000)
    product = 2 * product;
```
Counter-controlled repetition

- Loop repeated until counter reaches a certain value
- Definite repetition: number of repetitions is known
- Example: A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz

Pseudocode:

Set total to zero
Set grade counter to one
While grade counter is less than or equal to ten
   Input the next grade
   Add the grade into the total
   Add one to the grade counter
Set the class average to the total divided by ten
Print the class average
/ Fig. 3.6: fig03_06.c

Class average program with counter-controlled repetition */

#include <stdio.h>

/* function main begins program execution */

int main()
{
    int counter; /* number of grade to be entered next */
    int grade;  /* grade value */
    int total;  /* sum of grades input by user */
    int average; /* average of grades */

    /* initialization phase */
    total = 0;    /* initialize total */
    counter = 1; /* initialize loop counter */

    /* processing phase */
    while ( counter <= 10 ) { /* loop 10 times */
        printf( "Enter grade: " ); /* prompt for input */
        scanf( "%d", &grade ); /* read grade from user */
        total = total + grade; /* add grade to total */
        counter = counter + 1; /* increment counter */
    } /* end while */
/* termination phase */
average = total / 10; /* integer division */

/* display result */
printf( "Class average is %d\n", average );

return 0; /* indicate program ended successfully */

} /* end function main */
Nested control structures

Problem
- A college has a list of test results (1 = pass, 2 = fail) for 10 students
- Write a program that analyzes the results
  - If more than 8 students pass, print "Raise Tuition"

Notice that
- The program must process 10 test results
  - Counter-controlled loop will be used
- Two counters can be used
  - One for number of passes, one for number of fails
- Each test result is a number—either a 1 or a 2
  - If the number is not a 1, we assume that it is a 2
Nested control structures

- **Top level outline**
  Analyze exam results and decide if tuition should be raised

- **First Refinement**
  - Initialize variables
  - Input the ten quiz grades and count passes and failures
  - Print a summary of the exam results and decide if tuition should be raised

- **Refine Initialize variables to**
  - Initialize passes to zero
  - Initialize failures to zero
  - Initialize student counter to one
Refine \textit{Input the ten quiz grades and count passes and failures} to

While student counter is less than or equal to ten
Input the next exam result
If the student passed
Add one to passes
else
Add one to failures
Add one to student counter

Refine \textit{Print a summary of the exam results and decide if tuition should be raised} to

Print the number of passes
Print the number of failures
If more than eight students passed
Print “Raise tuition”
Nested control structures

Initialize passes to zero
Initialize failures to zero
Initialize student to one

While student counter is less than or equal to ten
  Input the next exam result

  If the student passed
    Add one to passes
  else
    Add one to failures

  Add one to student counter

Print the number of passes
Print the number of failures
If more than eight students passed
  Print “Raise tuition”
/* Fig. 3.10: fig03_10.c */
Analysis of examination results */
#include <stdio.h>

/* function main begins program execution */
int main()
{
    /* initialize variables in definitions */
    int passes = 0;  /* number of passes */
    int failures = 0; /* number of failures */
    int student = 1; /* student counter */
    int result;     /* one exam result */

    /* process 10 students using counter-controlled loop */
    while ( student <= 10 ) {

        /* prompt user for input and obtain value from user */
        printf( "Enter result ( 1=pass, 2=fail ): " );
        scanf( "%d", &result );

        /* if result 1, increment passes */
        if ( result == 1 ) {
            passes = passes + 1;
        } /* end if */
```c
else { /* otherwise, increment failures */
    failures = failures + 1;
} /* end else */

student = student + 1; /* increment student counter */
} /* end while */

/* termination phase; display number of passes and failures */
printf( "Passed %d\n", passes );
printf( "Failed %d\n", failures );

/* if more than eight students passed, print "raise tuition" */
if ( passes > 8 ) {
    printf( "Raise tuition\n" );
} /* end if */

return 0; /* indicate program ended successfully */
} /* end function main */
```
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Passed 6
Failed 4

Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Passed 9
Failed 1
Raise tuition
The Essentials of Repetition

- **Loop**
  - Group of instructions computer executes repeatedly while some condition remains true

- **Counter-controlled repetition**
  - Definite repetition: know how many times loop will execute
  - Control variable used to count repetitions

- **Sentinel-controlled repetition**
  - Indefinite repetition
  - Used when number of repetitions not known
  - Sentinel value indicates "end of data"
Counter-controlled repetition requires

- The name of a control variable (or loop counter)
- The initial value of the control variable
- An increment (or decrement) by which the control variable is modified each time through the loop
- A condition that tests for the final value of the control variable (i.e., whether looping should continue)
Example:

```c
int counter = 1;       // initialization
while ( counter <= 10 ) { // repetition condition
    printf( "%d\n", counter );
    ++counter;           // increment
}
```

- The statement
  ```c
  int counter = 1;
  ```
  - Names `counter`
  - Defines it to be an integer
  - Reserves space for it in memory
  - Sets it to an initial value of 1
Condensed code

- C Programmers would make the program more concise
- Initialize `counter` to 0
  
  ```c
  while ( ++counter <= 10 )
    printf( "%d\n, counter );
  ```
4.4 The **for** Repetition Statement

```
for ( counter = 1; counter <= 10; ++counter )
```

- **For** keyword
- Control variable name: `counter`
- Initial value of control variable: `1`
- Loop-continuation condition
- Increment of control variable: `++counter`
- Final value of control variable for which the condition is true: `<= 10`
The for Repetition Statement

- **Format when using for loops**

  ```c
  for ( initialization; loopContinuationTest; increment )
  statement
  ```

- **Example:**

  ```c
  for( int counter = 1; counter <= 10; counter ++ )
  printf( "%d\n", counter );
  ```

  Prints the integers from one to ten

  No semicolon (`;`) after last expression
For loops can usually be rewritten as while loops:

```
initialization;
while (loopContinuationTest) {
    statement;
    increment;
}
```

- **Initialization and increment**
  - Can be comma-separated lists
  - Example:
    ```
    for (int i = 0, j = 0; j + i <= 10; j++, i++)
        printf( "%d\n", j + i )
    ```
Arithmetic expressions

- Initialization, loop-continuation, and increment can contain arithmetic expressions. If $x$ equals 2 and $y$ equals 10

$$\text{for } (j = x; j <= 4 * x * y; j += y / x)$$

is equivalent to

$$\text{for } (j = 2; j <= 80; j += 5)$$

Notes about the for statement:

- "Increment" may be negative (decrement)
- If the loop continuation condition is initially false
  - The body of the for statement is not performed
  - Control proceeds with the next statement after the for statement
- Control variable
  - Often printed or used inside for body, but not necessary
The **for** Statement: Notes and Observations

- Establish *initial value* of control variable
  
  - `counter = 1`

- Determine if *final value* of control variable has been reached
  
  - `counter <= 10`

  **true**
  
  - `printf( "%d", counter );`

  **false**

  - Body of loop (this may be many statements)

- Increment the control variable
  
  - `counter++`
```c
/* Fig. 4.5: fig04_05.c */

#include <stdio.h>

/* function main begins program execution */
int main()
{
    int sum = 0; /* initialize sum */
    int number; /* number to be added to sum */

    for ( number = 2; number <= 100; number += 2 ) {
        sum += number; /* add number to sum */
    } /* end for */

    printf( "Sum is %d\n", sum ); /* output sum */

    return 0; /* indicate program ended successfully */
}
/* end function main */

Sum is 2550
```
/* Fig. 4.6: fig04_06.c
Calculating compound interest */
#include <stdio.h>
#include <math.h>

/* function main begins program execution */
int main()
{
  double amount; /* amount on deposit */
  double principal = 1000.0; /* starting principal */
  double rate = .05; /* interest rate */
  int year; /* year counter */

  /* output table column head */
  printf( "%4s%21s
", "Year", "Amount on deposit" );

  /* calculate amount on deposit for each of ten years */
  for ( year = 1; year <= 10; year++ )
  {
    /* calculate new amount for specified year */
    amount = principal * pow( 1.0 + rate, year );

    /* output one table row */
    printf( "%4d%21.2f\n", year, amount );
  }

  return 0;
}
The switch Multiple-Selection Statement

- **switch**
  - Useful when a variable or expression is tested for all the values it can assume and different actions are taken

- **Format**
  - Series of `case` labels and an optional `default` case
    ```java
    switch (value) {
    case '1':
        actions
    case '2':
        actions
    default:
        actions
    }
    ```
  - `break;` exits from statement
Flowchart of the `switch` statement

- case a
  - true: case a action(s) -> break
  - false: false

- case b
  - true: case b action(s) -> break
  - false: false

- case z
  - true: case z action(s) -> break
  - false: false

- default action(s)
/* Fig. 4.7: fig04_07.c
Counting letter grades */
#include <stdio.h>

/* function main begins program execution */
int main()
{
    int grade;  /* one grade */
    int aCount = 0; /* number of As */
    int bCount = 0; /* number of Bs */
    int cCount = 0; /* number of Cs */
    int dCount = 0; /* number of Ds */
    int fCount = 0; /* number of Fs */

    printf(  "Enter the letter grades.\n"  );
    printf(  "Enter the EOF character to end input.\n"   );

    /* loop until user types end-of-file key sequence */
    while ( ( grade = getchar() ) != EOF ) {

        /* determine which grade was input */
        switch ( grade ) { /* switch nested in while */

            case 'A':  /* grade was uppercase A */
            case 'a':  /* or lowercase a */
              ++aCount; /* increment aCount */
              break;    /* necessary to exit switch */

            case 'B':  /* grade was uppercase B */
            case 'b':  /* or lowercase b */
              ++bCount; /* increment bCount */
              break;    /* necessary to exit switch */

            case 'C':  /* grade was uppercase C */
            case 'c':  /* or lowercase c */
              ++cCount; /* increment cCount */
              break;    /* necessary to exit switch */

            case 'D':  /* grade was uppercase D */
            case 'd':  /* or lowercase d */
              ++dCount; /* increment dCount */
              break;    /* necessary to exit switch */

            case 'F':  /* grade was uppercase F */
            case 'f':  /* or lowercase f */
              ++fCount; /* increment fCount */
              break;    /* necessary to exit switch */

            default:
              /* grade was not an A, B, C, D, or F */
              printf(  "Invalid grade entered.\n"  );

        }
    }

    /* display grade counts */
    printf(  "A's: %d\n", aCount  );
    printf(  "B's: %d\n", bCount  );
    printf(  "C's: %d\n", cCount  );
    printf(  "D's: %d\n", dCount  );
    printf(  "F's: %d\n", fCount  );

    return 0; /* indicates successful completion */
}
case 'B':    /* grade was uppercase B */
case 'b':    /* or lowercase b */
    ++bCount; /* increment bCount */
    break;    /* exit switch */

case 'C':    /* grade was uppercase C */
case 'c':    /* or lowercase c */
    ++cCount; /* increment cCount */
    break;    /* exit switch */

case 'D':    /* grade was uppercase D */
case 'd':    /* or lowercase d */
    ++dCount; /* increment dCount */
    break;    /* exit switch */

case 'F':    /* grade was uppercase F */
case 'f':    /* or lowercase f */
    ++fCount; /* increment fCount */
    break;    /* exit switch */

case '\n':   /* ignore newlines, */
case '\t':   /* tabs, */
case ' ':    /* and spaces in input */
    break;    /* exit switch */
default: /* catch all other characters */
    printf( "Incorrect letter grade entered.\n" );
    printf( " Enter a new grade.\n" );
    break; /* optional; will exit switch anyway */
} /* end switch */
} /* end while */

/* output summary of results */
printf( "\nTotals for each letter grade are:\n" );
printf( "A: %d\n", aCount ); /* display number of A grades */
printf( "B: %d\n", bCount ); /* display number of B grades */
printf( "C: %d\n", cCount ); /* display number of C grades */
printf( "D: %d\n", dCount ); /* display number of D grades */
printf( "F: %d\n", fCount ); /* display number of F grades */

return 0; /* indicate program ended successfully */
} /* end function main */
Enter the letter grades.
Enter the EOF character to end input.

a
b
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
~Z

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
The do...while Repetition Statement

- **The do...while repetition statement**
  - Similar to the while structure
  - Condition for repetition tested after the body of the loop is performed
    - All actions are performed at least once
  - Format:
    ```
    do {
        statement;
    } while ( condition );
    ```
The `do...while` Repetition Statement

- Example (letting counter = 1):
  
  ```c
  do {
      printf( "%d ", counter );
  } while (++counter <= 10);
  ```

  - Prints the integers from 1 to 10
Flowchart of the do...while repetition statement
/* Fig. 4.9: fig04_09.c */

#include <stdio.h>

/* function main begins program execution */

int main()
{
    int counter = 1; /* initialize counter */

do {
    printf( "%d  ", counter ); /* display counter */
} while ( ++counter <= 10 ); /* end do...while */

return 0; /* indicate program ended successfully */

} /* end function main */
The break and continue Statements

- **break**
  - Causes immediate exit from a `while`, `for`, `do...while` or `switch` statement
  - Program execution continues with the first statement after the structure
  - Common uses of the `break` statement
    - Escape early from a loop
    - Skip the remainder of a `switch` statement
/* Fig. 4.11: fig04_11.c */

#include <stdio.h>

/* function main begins program execution */
int main()
{
    int x; /* counter */

    /* loop 10 times */
    for ( x = 1; x <= 10; x++ ) {
        /* if x is 5, terminate loop */
        if ( x == 5 ) {
            break; /* break loop only if x is 5 */
        } /* end if */

        printf( "%d", x ); /* display value of x */
    } /* end for */

    printf( "\nBroke out of loop at x == %d", x );

    return 0; /* indicate program ended successfully */
} /* end function main */

Broke out of loop at x == 5
The *break* and *continue* Statements

- **continue**
  - Skips the remaining statements in the body of a `while`, `for` or `do...while` statement
    - Proceeds with the next iteration of the loop
  - `while` and `do...while`
    - Loop-continuation test is evaluated immediately after the `continue` statement is executed
  - `for`
    - Increment expression is executed, then the loop-continuation test is evaluated
/* Fig. 4.12: fig04_12.c
Using the continue statement in a for statement */

#include <stdio.h>

/* function main begins program execution */
int main()
{
    int x; /* counter */

    /* loop 10 times */
    for ( x = 1; x <= 10; x++ ) {

        /* if x is 5, continue with next iteration of loop */
        if ( x == 5 ) {
            continue; /* skip remaining code in loop body */
        } /* end if */

        printf( "%d ", x ); /* display value of x */
    } /* end for */

    printf( "\nUsed continue to skip printing the value 5\n" );

    return 0; /* indicate program ended successfully */
} /* end function main */

Used continue to skip printing the value 5
Logical Operators

- **&& (logical AND)**
  - Returns `true` if both conditions are `true`

- **|| (logical OR)**
  - Returns `true` if either of its conditions are `true`

- **! (logical NOT, logical negation)**
  - Reverses the truth/falsity of its condition
  - Unary operator, has one operand

**Useful as conditions in loops**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>true &amp;&amp; false</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td>`true</td>
<td></td>
</tr>
<tr>
<td><code>!false</code></td>
<td><code>true</code></td>
</tr>
</tbody>
</table>
## Logical Operators

### Logical AND ($\&\&$)

<table>
<thead>
<tr>
<th>expression1</th>
<th>expression2</th>
<th>expression1 $&amp;&amp;$ expression2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>0</td>
</tr>
<tr>
<td>nonzero</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nonzero</td>
<td>nonzero</td>
<td>1</td>
</tr>
</tbody>
</table>

*Fig. 4.13* Truth table for the $\&\&$ (logical AND) operator.

### Logical OR (||)

| expression1 | expression2 | expression1 || expression2 |
|-------------|-------------|----------------|
| 0           | 0           | 0              |
| 0           | nonzero     | 1              |
| nonzero     | 0           | 1              |
| nonzero     | nonzero     | 1              |

*Fig. 4.14* Truth table for the logical OR (||) operator.

### Logical Negation (!)

<table>
<thead>
<tr>
<th>expression</th>
<th>! expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>nonzero</td>
<td>0</td>
</tr>
</tbody>
</table>

*Fig. 4.15* Truth table for operator `!` (logical negation).
<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>++</code></td>
<td>right to left</td>
<td>unary</td>
</tr>
<tr>
<td><code>-</code></td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td><code>+</code></td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td><code>=</code></td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>left to right</td>
<td>logical AND</td>
</tr>
<tr>
<td>`</td>
<td></td>
<td>`</td>
</tr>
<tr>
<td><code>?:</code></td>
<td>right to left</td>
<td>conditional</td>
</tr>
<tr>
<td><code>+=</code></td>
<td>right to left</td>
<td>assignment</td>
</tr>
<tr>
<td><code>-=</code></td>
<td>right to left</td>
<td>assignment</td>
</tr>
<tr>
<td><code>*=</code></td>
<td>right to left</td>
<td>assignment</td>
</tr>
<tr>
<td><code>/=</code></td>
<td>right to left</td>
<td>assignment</td>
</tr>
<tr>
<td><code>%=</code></td>
<td>right to left</td>
<td>assignment</td>
</tr>
<tr>
<td><code>=</code></td>
<td>left to right</td>
<td>comma</td>
</tr>
</tbody>
</table>

Fig. 4.16 Operator precedence and associativity.
Dangerous error
- Does not ordinarily cause syntax errors
- Any expression that produces a value can be used in control structures
- Nonzero values are true, zero values are false
- Example using ==:
  ```c
  if ( payCode == 4 )
    printf( "You get a bonus!\n" );
  ```
  - Checks payCode, if it is 4 then a bonus is awarded
Example, replacing == with =:

```c
if ( payCode = 4 )
    printf( "You get a bonus!\n" );
```

- This sets `payCode` to 4
- 4 is nonzero, so expression is `true`, and bonus awarded no matter what the `payCode` was

Logic error, not a syntax error
Confusing Equality (==) and Assignment (=) Operators

- **lvalues**
  - Expressions that can appear on the left side of an equation
  - Their values can be changed, such as variable names
    - \( x = 4; \)

- **rvalues**
  - Expressions that can only appear on the right side of an equation
  - Constants, such as numbers
    - Cannot write \( 4 = x; \)
    - Must write \( x = 4; \)
  - lvalues can be used as rvalues, but not vice versa
    - \( y = x; \)
Structured-Programming

Sequence

if statement (single selection)

if...else statement (double selection)

switch statement (multiple selection)

break
Structured-Programming

Repetition

while statement

T

F

F

do..while statement

T

F

for statement

T

F
Structured programming
- Easier than unstructured programs to understand, test, debug and, modify programs

Rules for structured programming
- Rules developed by programming community
- Only single-entry/single-exit control structures are used
- Rules:
  1. Begin with the “simplest flowchart”
  2. Stacking rule: Any rectangle (action) can be replaced by two rectangles (actions) in sequence
  3. Nesting rule: Any rectangle (action) can be replaced by any control structure (sequence, if, if...else, switch, while, do...while or for)
  4. Rules 2 and 3 can be applied in any order and multiple times
Structured-Programming Summary

Rule 1 - Begin with the simplest flowchart

Rule 2 - Any rectangle can be replaced by two rectangles in sequence
Structured-Programming Summary

Rule 3 - Replace any rectangle with a control structure
Structured-Programming Summary

- Stacked building blocks
- Nested building blocks
- Overlapping building blocks (illegal in structured programs)
Figure 4.23 An unstructured flowchart.
Structured-Programming Summary

All programs can be broken down into 3 controls

- **Sequence** – handled automatically by compiler
- **Selection** – `if`, `if...else` or `switch`
- **Repetition** – `while`, `do...while` or `for`
  - Can only be combined in two ways
    - Nesting (rule 3)
    - Stacking (rule 2)

Any selection can be rewritten as an `if` statement, and any repetition can be rewritten as a `while` statement
Program Modules in C

- **Functions**
  - Modules in C
  - Programs combine user-defined functions with library functions
    - C standard library has a wide variety of functions

- **Function calls**
  - Invoking functions
    - Provide function name and arguments (data)
    - Function performs operations or manipulations
    - Function returns results
  - Function call analogy:
    - Boss asks worker to complete task
      - Worker gets information, does task, returns result
      - Information hiding: boss does not know details
Fig. 5.1 Hierarchical boss function/worker function relationship.
Math Library Functions

- **Math library functions**
  - perform common mathematical calculations
  - `#include <math.h>`

- **Format for calling functions**
  - `FunctionName( argument );`
    - If multiple arguments, use comma-separated list
  - `printf( "%.2f", sqrt( 900.0 ) );`
    - Calls function `sqrt`, which returns the square root of its argument
    - All math functions return data type `double`
  - Arguments may be constants, variables, or expressions
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sqrt(x)</code></td>
<td>square root of x</td>
<td><code>sqrt(900.0)</code> is 30.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>sqrt(9.0)</code> is 3.0</td>
</tr>
<tr>
<td><code>exp(x)</code></td>
<td>exponential function $e^x$</td>
<td><code>exp(1.0)</code> is 2.718282</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>exp(2.0)</code> is 7.389056</td>
</tr>
<tr>
<td><code>log(x)</code></td>
<td>natural logarithm of x (base $e$)</td>
<td><code>log(2.718282)</code> is 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>log(7.389056)</code> is 2.0</td>
</tr>
<tr>
<td><code>log10(x)</code></td>
<td>logarithm of x (base 10)</td>
<td><code>log10(1.0)</code> is 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>log10(10.0)</code> is 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>log10(100.0)</code> is 2.0</td>
</tr>
<tr>
<td><code>fabs(x)</code></td>
<td>absolute value of x</td>
<td><code>fabs(5.0)</code> is 5.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>fabs(0.0)</code> is 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>fabs(-5.0)</code> is 5.0</td>
</tr>
<tr>
<td><code>ceil(x)</code></td>
<td>rounds x to the smallest integer not less than x</td>
<td><code>ceil(9.2)</code> is 10.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>ceil(-9.8)</code> is -9.0</td>
</tr>
<tr>
<td><code>floor(x)</code></td>
<td>rounds x to the largest integer not greater than x</td>
<td><code>floor(9.2)</code> is 9.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>floor(-9.8)</code> is -10.0</td>
</tr>
<tr>
<td><code>pow(x, y)</code></td>
<td>$x$ raised to power $y$ ($x^y$)</td>
<td><code>pow(2, 7)</code> is 128.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>pow(9, 0.5)</code> is 3.0</td>
</tr>
<tr>
<td><code>fmod(x, y)</code></td>
<td>remainder of $x/y$ as a floating point number</td>
<td><code>fmod(13.657, 2.333)</code> is 1.992</td>
</tr>
<tr>
<td><code>sin(x)</code></td>
<td>trigonometric sine of $x$ (x in radians)</td>
<td><code>sin(0.0)</code> is 0.0</td>
</tr>
<tr>
<td><code>cos(x)</code></td>
<td>trigonometric cosine of $x$ (x in radians)</td>
<td><code>cos(0.0)</code> is 1.0</td>
</tr>
<tr>
<td><code>tan(x)</code></td>
<td>trigonometric tangent of $x$ (x in radians)</td>
<td><code>tan(0.0)</code> is 0.0</td>
</tr>
</tbody>
</table>

Fig. 5.2 Commonly used math library functions.
Functions

- Modularize a program
- All variables defined inside functions are local variables
  - Known only in function defined
- Parameters
  - Communicate information between functions
  - Local variables

Benefits of functions

- Divide and conquer
  - Manageable program development
- Software reusability
  - Use existing functions as building blocks for new programs
  - Abstraction - hide internal details (library functions)
- Avoid code repetition
Function definition format

```
return-value-type  function-name( parameter-list )
{
    declarations and statements
}
```

- **Function-name**: any valid identifier
- **Return-value-type**: data type of the result (default `int`)
  - `void` – indicates that the function returns nothing
- **Parameter-list**: comma separated list, declares parameters
  - A type must be listed explicitly for each parameter unless, the parameter is of type `int`
Function Definitions

Function definition format (continued)

```
return-value-type function-name( parameter-list )
{
    declarations and statements
}
```

- Definitions and statements: function body (block)
  - Variables can be defined inside blocks (can be nested)
  - Functions can not be defined inside other functions

- Returning control
  - If nothing returned
    - `return;`
    - or, until reaches right brace
  - If something returned
    - `return expression;`
/* Fig. 5.3: fig05_03.c  
    Creating and using a programmer-defined function */
#include <stdio.h>

int square( int y ); /* function prototype */

/* function main begins program execution */
int main()
{
    int x; /* counter */

    /* loop 10 times and calculate and output square of x each time */
    for ( x = 1; x <= 10; x++ ) {
        printf( "%d , square( x ) ); /* function call */
    } /* end for */

    printf( "\n" );

    return 0; /* indicates successful termination */
} /* end main */
/* square function definition returns square of an integer */
int square( int y ) /* y is a copy of argument to function */
{
    return y * y; /* returns square of y as an int */
} /* end function square */

1  4  9  16  25  36  49  64  81  100
/* Fig. 5.4: fig05_04.c 
Finding the maximum of three integers */
#include <stdio.h>

int maximum( int x, int y, int z ); /* function prototype */

/* function main begins program execution */
int main()
{
    int number1; /* first integer */
    int number2; /* second integer */
    int number3; /* third integer */

    printf( "Enter three integers: " );
    scanf( "%d%d%d", &number1, &number2, &number3 );

    /* number1, number2 and number3 are arguments 
    to the maximum function call */
    printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );

    return 0; /* indicates successful termination */
} /* end main */
/* Function maximum definition */
/* x, y and z are parameters */
int maximum(int x, int y, int z)
{
    int max = x; /* assume x is largest */

    if (y > max) { /* if y is larger than max, assign y to max */
        max = y;
    } /* end if */

    if (z > max) { /* if z is larger than max, assign z to max */
        max = z;
    } /* end if */

    return max; /* max is largest value */
} /* end function maximum */

Enter three integers: 22 85 17
Maximum is: 85
Enter three integers: 85 22 17
Maximum is: 85
Enter three integers: 22 17 85
Maximum is: 85
Function Prototypes

- **Function prototype**
  - Function name
  - Parameters – what the function takes in
  - Return type – data type function returns (default `int`)
  - Used to validate functions
  - Prototype only needed if function definition comes after use in program
  - The function with the prototype
    ```
    int maximum ( int x, int y, int z );
    ```
    - Takes in 3 `int`
    - Returns an `int`

- **Promotion rules and conversions**
  - Converting to lower types can lead to errors
## Function Prototypes

<table>
<thead>
<tr>
<th>Data types</th>
<th>printf conversion specifications</th>
<th>scanf conversion specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>long double</td>
<td>%Lf</td>
<td>%Lf</td>
</tr>
<tr>
<td>double</td>
<td>%f</td>
<td>%f</td>
</tr>
<tr>
<td>float</td>
<td>%f</td>
<td>%f</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>%lu</td>
<td>%u</td>
</tr>
<tr>
<td>long int</td>
<td>%d</td>
<td>%d</td>
</tr>
<tr>
<td>unsigned int</td>
<td>%u</td>
<td>%u</td>
</tr>
<tr>
<td>int</td>
<td>%d</td>
<td>%d</td>
</tr>
<tr>
<td>short</td>
<td>%hd</td>
<td>%hd</td>
</tr>
<tr>
<td>char</td>
<td>%c</td>
<td>%c</td>
</tr>
</tbody>
</table>

Fig. 5.5 Promotion hierarchy for data types.
**Pointer variables**

- Contain memory addresses as their values
- Normal variables contain a specific value (direct reference)

- Pointers contain address of a variable that has a specific value (indirect reference)
- Indirection – referencing a pointer value
**Pointer definitions**

- * used with pointer variables
  
  ```c
  int *myPtr;
  ```
- Defines a pointer to an `int` (pointer of type `int *`)
- Multiple pointers require using a * before each variable definition
  
  ```c
  int *myPtr1, *myPtr2;
  ```
- Can define pointers to any data type
- Initialize pointers to 0, `NULL`, or an address
  - 0 or `NULL` – points to nothing (`NULL` preferred)
**& (address operator)**

- Returns address of operand

```c
int y = 5;
int *yPtr;
yPtr = &y; /**< yPtr gets address of y */
yPtr "points to" y
```

Address of `y` is value of `yPtr`
Pointer Operators

- ***(indirection/dereferencing operator)***
  - Returns a synonym/alias of what its operand points to
  - \(*_{yptr}\) returns \(y\) (because \(yptr\) points to \(y\))
  - \(\ast\) can be used for assignment
    - Returns alias to an object
      
      \[
      \ast_{yptr} = 7; /* changes y to 7 */
      \]
  - Dereferenced pointer (operand of \(\ast\)) must be an lvalue (no constants)

- **\(*\ and \& are inverses***
  - They cancel each other out
**Call by reference with pointer arguments**

- Pass address of argument using `&` operator
- Allows you to change actual location in memory
- Arrays are not passed with `&` because the array name is already a pointer

**`*` operator**

- Used as alias/nickname for variable inside of function

```c
void double( int *number )
{
    *number = 2 * ( *number );
}
```

- `*number` used as nickname for the variable passed
/* Fig. 7.6: fig07_06.c 
Cube a variable using call-by-value */

#include <stdio.h>

int cubeByValue( int n ); /* prototype */

int main()
{
    int number = 5; /* initialize number */

    printf( "The original value of number is %d", number );

    /* pass number by value to cubeByValue */
    number = cubeByValue( number );

    printf( "\nThe new value of number is %d\n", number );

    return 0; /* indicates successful termination */
}

/* calculate and return cube of integer argument */
int cubeByValue( int n )
{
    return n * n * n; /* cube local variable n and return result */
}

} /* end function cubeByValue */
Arithmetic operations can be performed on pointers

- Increment/decrement pointer (\texttt{++} or \texttt{- -})
- Add an integer to a pointer (\texttt{+} or \texttt{+=}, \texttt{-} or \texttt{-=})
- Pointers may be subtracted from each other
- Operations meaningless unless performed on an array
5 element int array on machine with 4 byte ints

- vPtr points to first element v[0]
  - at location 3000 (vPtr = 3000)
- vPtr += 2; sets vPtr to 3008
  - vPtr points to v[2] (incremented by 2), but the machine has 4 byte ints, so it points to address 3008
Subtracting pointers

- Returns number of elements from one to the other. If
  \[
  \text{vPtr2} = \text{v[2]}; \\
  \text{vPtr} = \text{v[0]}; \\
  \]
  \[
  \text{vPtr2} - \text{vPtr}
  \]
  would produce 2

Pointer comparison (\(<\),\(=\),\(>\))

- See which pointer points to the higher numbered array element
- Also, see if a pointer points to 0
Pointers of the same type can be assigned to each other

- If not the same type, a cast operator must be used
- Exception: pointer to `void` (type `void *`)
  - Generic pointer, represents any type
  - No casting needed to convert a pointer to `void` pointer
  - `void` pointers cannot be dereferenced
Arrays and pointers closely related

- Array name like a constant pointer
- Pointers can do array subscripting operations

Define an array `b[5]` and a pointer `bPtr`

- To set them equal to one another use:
  ```
  bPtr = b;
  ```
- The array name (`b`) is actually the address of first element of the array `b[5]`
  ```
  bPtr = &b[0]
  ```
- Explicitly assigns `bPtr` to address of first element of `b`
The Relationship Between Pointers and Arrays

- **Element b[ 3 ]**
  - Can be accessed by *( bPtr + 3 )
    - Where n is the offset. Called pointer/offset notation
  - Can be accessed by bPtr[ 3 ]
    - Called pointer/subscript notation
    - bPtr[ 3 ] same as b[ 3 ]
  - Can be accessed by performing pointer arithmetic on the array itself
    - *( b + 3 )
/* Fig. 7.20: fig07_20.cpp */

#include <stdio.h>

int main()
{
    int b[] = { 10, 20, 30, 40 }; /* initialize array b */
    int *bPtr = b;                /* set bPtr to point to array b */
    int i;                        /* counter */
    int offset;                   /* counter */

    /* output array b using array subscript notation */
    printf( "Array b printed with:
Array subscript notation\n" );

    /* loop through array b */
    for ( i = 0; i < 4; i++ ) {
        printf( "b[ %d ] = %d\n", i, b[ i ] );
    } /* end for */

    /* output array b using array name and pointer/offset notation */
    printf( "\nPointer/offset notation where
    "the pointer is the array name\n" );
/* loop through array b */
for ( offset = 0; offset < 4; offset++ ) {
    printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
} /* end for */

/* output array b using bPtr and array subscript notation */
printf( "\nPointer subscript notation\n" );

/* loop through array b */
for ( i = 0; i < 4; i++ ) {
    printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ] );
} /* end for */

/* output array b using bPtr and pointer/offset notation */
printf( "\nPointer/offset notation\n" );

/* loop through array b */
for ( offset = 0; offset < 4; offset++ ) {
    printf( "*( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );
} /* end for */

return 0; /* indicates successful termination */

} /* end main */
/ * Fig. 7.21: fig07_21.c */
#include <stdio.h>

void copy1( char *s1, const char *s2 ); /* prototype */
void copy2( char *s1, const char *s2 ); /* prototype */

int main()
{
  char string1[10];          /* create array string1 */
  char *string2 = "Hello";   /* create a pointer to a string */
  char string3[10];          /* create array string3 */
  char string4[] = "Good Bye"; /* create a pointer to a string */

  copy1( string1, string2 );
  printf("string1 = %s\n", string1 );

  copy2( string3, string4 );
  printf("string3 = %s\n", string3 );

  return 0; /* indicates successful termination */
} /* end main */
```c
/* copy s2 to s1 using array notation */
void copy1( char *s1, const char *s2 )
{
    int i; /* counter */

    /* loop through strings */
    for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ ) {
        ; /* do nothing in body */
    } /* end for */

} /* end function copy1 */

/* copy s2 to s1 using pointer notation */
void copy2( char *s1, const char *s2 )
{
    /* loop through strings */
    for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ ) {
        ; /* do nothing in body */
    } /* end for */

} /* end function copy2 */

string1 = Hello
string3 = Good Bye
```
Arrays of Pointers

- Arrays can contain pointers
- For example: an array of strings

```c
```

- Strings are pointers to the first character
- `char *` — each element of `suit` is a pointer to a `char`
- The strings are not actually stored in the array `suit`, only pointers to the strings are stored

- `suit` array has a fixed size, but strings can be of any size
Card shuffling program

- Use array of pointers to strings
- Use double scripted array (suit, face)

The numbers 1-52 go into the array
- Representing the order in which the cards are dealt
Pseudocode

- **Top level:**
  
  *Shuffle and deal 52 cards*

- **First refinement:**
  
  *Initialize the suit array*
  
  *Initialize the face array*
  
  *Initialize the deck array*
  
  *Shuffle the deck*
  
  *Deal 52 cards*
Second refinement

• Convert *shuffle the deck* to
  
  *For each of the 52 cards*
  
  *Place card number in randomly selected unoccupied slot of deck*

• Convert *deal 52 cards* to
  
  *For each of the 52 cards*
  
  *Find card number in deck array and print face and suit of card*
• Convert *shuffle the deck* to
  
  \begin{align*}
  \text{Choose slot of deck randomly} \\
  \text{While chosen slot of deck has been previously chosen} \\
  \text{Choose slot of deck randomly} \\
  \text{Place card number in chosen slot of deck}
  \end{align*}

• Convert *deal 52 cards* to
  
  \begin{align*}
  \text{For each slot of the deck array} \\
  \text{If slot contains card number} \\
  \text{Print the face and suit of the card}
  \end{align*}
/* Fig. 7.24: fig07_24.c
Card shuffling dealing program*/

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

/* prototypes */
void shuffle( int wDeck[][ 13 ] );
void deal( const int wDeck[][ 13 ], const char *wFace[], const char *wSuit[] );

int main()
{
    /* initialize suit array */

    /* initialize face array */
    const char *face[ 13 ] =

    /* initialize deck array */
    int deck[ 4 ][ 13 ] = { 0 };
srand( time( 0 ) ); /* seed random number generator */

shuffle( deck );
deal( deck, face, suit );

return 0; /* indicates successful termination */

} /* end main */

/* shuffle cards in deck */
void shuffle( int wDeck[][ 13 ] )
{
    int row;    /* row number */
    int column; /* column number */
    int card;   /* counter */

    /* for each of the 52 cards, choose slot of deck randomly */
    for ( card = 1; card <= 52; card++ ) {

        /* choose new random location until unoccupied slot found */
        do {
            row = rand() % 4;
            column = rand() % 13;
        } while( wDeck[ row ][ column ] != 0 ); /* end do...while */
/ * place card number in chosen slot of deck */

    wDeck[ row ][ column ] = card;
} /* end for */

} /* end function shuffle */

/* deal cards in deck */

void deal( const int wDeck[ ] [ 13 ], const char *wFace[ ],
            const char *wSuit[ ] )
{

    int card;    /* card counter */
    int row;     /* row counter */
    int column;  /* column counter */

    /* deal each of the 52 cards */
    for ( card = 1; card <= 52; card++ ) {

        /* loop through rows of wDeck */
        for ( row = 0; row <= 3; row++ ) {

            /* loop through columns of wDeck for current row */
            for ( column = 0; column <= 12; column++ ) {

                /* if slot contains current card, display card */
                if ( wDeck[ row ][ column ] == card ) {
printf("%s of %8s", wFace[ column ], wSuit[ row ],
    card % 2 == 0 ? '\n' : '\t');
} /* end if */

} /* end for */

} /* end for */

} /* end for */

} /* end function deal */
Fundamentals of Strings and Characters

- **Characters**
  - Building blocks of programs
    - Every program is a sequence of meaningfully grouped characters
  - Character constant
    - An int value represented as a character in single quotes
    - `'z'` represents the integer value of `z`

- **Strings**
  - Series of characters treated as a single unit
    - Can include letters, digits and special characters (`*, /, $`)
  - String literal (string constant) - written in double quotes
    - "Hello"
  - Strings are arrays of characters
    - String a pointer to first character
    - Value of string is the address of first character
Fundamentals of Strings and Characters

❖ **String definitions**
  - Define as a character array or a variable of type `char *`
    ```
    char color[] = "blue";
    char *colorPtr = "blue";
    ```
  - Remember that strings represented as character arrays end with `\0`
    - `color` has 5 elements

❖ **Inputting strings**
  - Use `scanf`
    ```
    scanf("%s", word);
    ```
  - Copies input into `word[]`
  - Do not need `&` (because a string is a pointer)
    - Remember to leave room in the array for `\0`
Character Handling Library

- Character handling library
  - Includes functions to perform useful tests and manipulations of character data
  - Each function receives a character (an int) or EOF as an argument
- The following slide contains a table of all the functions in <ctype.h>
## 8.3 Character Handling Library

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int isdigit(int c);</td>
<td>Returns true if c is a digit and false otherwise.</td>
</tr>
<tr>
<td>int isalpha(int c);</td>
<td>Returns true if c is a letter and false otherwise.</td>
</tr>
<tr>
<td>int isalnum(int c);</td>
<td>Returns true if c is a digit or a letter and false otherwise.</td>
</tr>
<tr>
<td>int issxdigit(int c);</td>
<td>Returns true if c is a hexadecimal digit character and false otherwise.</td>
</tr>
<tr>
<td>int islower(int c);</td>
<td>Returns true if c is a lowercase letter and false otherwise.</td>
</tr>
<tr>
<td>int isupper(int c);</td>
<td>Returns true if c is an uppercase letter; false otherwise.</td>
</tr>
<tr>
<td>int tolower(int c);</td>
<td>If c is an uppercase letter, tolower returns c as a lowercase letter. Otherwise, tolower returns the argument unchanged.</td>
</tr>
<tr>
<td>int toupper(int c);</td>
<td>If c is a lowercase letter, toupper returns c as an uppercase letter. Otherwise, toupper returns the argument unchanged.</td>
</tr>
<tr>
<td>int isspace(int c);</td>
<td>Returns true if c is a white-space character—newline (\n), space (\space), form feed (\r\f), carriage return (\r\n), horizontal tab (\t), or vertical tab (\v)—and false otherwise</td>
</tr>
<tr>
<td>int iscntrl(int c);</td>
<td>Returns true if c is a control character and false otherwise.</td>
</tr>
<tr>
<td>int ispunct(int c);</td>
<td>Returns true if c is a printing character other than a space, a digit, or a letter and false otherwise.</td>
</tr>
<tr>
<td>int isprint(int c);</td>
<td>Returns true value if c is a printing character including space (\space) and false otherwise.</td>
</tr>
<tr>
<td>int isgraph(int c);</td>
<td>Returns true if c is a printing character other than space (\space) and false otherwise.</td>
</tr>
</tbody>
</table>
/* Fig. 8.2: fig08_02.c
Using functions isdigit, isalpha, isalnum, and isxdigit */

#include <stdio.h>
#include <ctype.h>

int main()
{
printf( "According to isdigit: ",
    isdigit( '8' ) ? "8 is a " : "8 is not a ", "digit",
    isdigit( '#' ) ? "# is a " : "# is not a ", "digit" );

printf( "According to isalpha:",
    isalpha( 'A' ) ? "A is a " : "A is not a ", "letter",
    isalpha( 'b' ) ? "b is a " : "b is not a ", "letter",
    isalpha( '&' ) ? ",& is a " : ",& is not a ", "letter",
    isalpha( '4' ) ? "4 is a " : "4 is not a ", "letter" );

printf( "According to isalnum",
    isalnum( 'A' ) ? "A is a " : "A is not a ", "digit or a letter",
    isalnum( '8' ) ? "8 is a " : "8 is not a ", "digit or a letter",
    isalnum( '#' ) ? "# is a " : "# is not a ", "digit or a letter" );
}
printf("According to isxdigit:
\tisxdigit('F') ? "F is a " : "F is not a 
\t"hexadecimal digit",
\tisxdigit('J') ? "J is a " : "J is not a 
\t"hexadecimal digit",
\tisxdigit('7') ? "7 is a " : "7 is not a 
\t"hexadecimal digit",
\tisxdigit('$') ? "$ is a " : "$ is not a 
\t"hexadecimal digit",
\tisxdigit('f') ? "f is a " : "f is not a 
\t"hexadecimal digit" );

return 0; /* indicates successful termination */
*/
/* Fig. 8.3: fig08_03.c */

#include <stdio.h>
#include <ctype.h>

int main()
{
    printf( "According to islower:
            islower('p') ? "p is a " : "p is not a ",
            "lowercase letter",
            islower('P') ? "P is a " : "P is not a ",
            "lowercase letter",
            islower('5') ? "5 is a " : "5 is not a ",
            "lowercase letter",
            islower('!' ) ? "! is a " : "! is not a ",
            "lowercase letter" );

    printf( "According to isupper:
            isupper('D') ? "D is an " : "D is not an ",
            "uppercase letter",
            isupper('d') ? "d is an " : "d is not an ",
            "uppercase letter",
            isupper('8') ? "8 is an " : "8 is not an ",
            "uppercase letter",
            isupper('$') ? "$ is an " : "$ is not an ",
            "uppercase letter" );
}
According to islower:
- p is a lowercase letter
- P is not a lowercase letter
- 5 is not a lowercase letter
- ! is not a lowercase letter

According to isupper:
- D is an uppercase letter
- d is not an uppercase letter
- 8 is not an uppercase letter
- $ is not an uppercase letter

- u converted to uppercase is U
- 7 converted to uppercase is 7
- $ converted to uppercase is $
- L converted to lowercase is l
String Conversion Functions

- Conversion functions
  - In `<stdlib.h>` (general utilities library)
  - Convert strings of digits to integer and floating-point values

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double atof(const char *nPtr);</code></td>
<td>Converts the string <code>nPtr</code> to <code>double</code>.</td>
</tr>
<tr>
<td><code>int atoi(const char *nPtr);</code></td>
<td>Converts the string <code>nPtr</code> to <code>int</code>.</td>
</tr>
<tr>
<td><code>long atol(const char *nPtr);</code></td>
<td>Converts the string <code>nPtr</code> to <code>long int</code>.</td>
</tr>
<tr>
<td><code>double strtod(const char *nPtr, char **endPtr);</code></td>
<td>Converts the string <code>nPtr</code> to <code>double</code>.</td>
</tr>
<tr>
<td><code>long strtol(const char *nPtr, char **endPtr, int base);</code></td>
<td>Converts the string <code>nPtr</code> to <code>long</code>.</td>
</tr>
<tr>
<td><code>unsigned long strtoul(const char *nPtr, char **endPtr, int base);</code></td>
<td>Converts the string <code>nPtr</code> to <code>unsigned long</code>.</td>
</tr>
</tbody>
</table>
/* Fig. 8.6: fig08_06.c  
 * Using atof */

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

int main()
{

double d; /* variable to hold converted string */

d = atof("99.0");

printf("The string "99.0" converted to double is ", d,  
"The converted value divided by 2 is ",  
d / 2.0);

return 0; /* indicates successful termination */
}

The string "99.0" converted to double is 99.000
The converted value divided by 2 is 49.500
/* Fig. 8.7: fig08_07.c  
Using atoi */

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

int main()
{
    int i; /* variable to hold converted string */

    i = atoi( "2593" );

    printf( "%s
" ,  "The string "2593" converted to int is ", i ,  "The converted value minus 593 is ", i - 593);

    return 0; /* indicates successful termination */
}

The string "2593" converted to int is 2593
The converted value minus 593 is 2000
/* Fig. 8.8: fig08_08.c */

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

int main()
{
    long l; /* variable to hold converted string */

    l = atol( "1000000" );

    printf( "The string \"1000000\" converted to long int is \", l,
            "The converted value divided by 2 is \", l / 2 );

    return 0; /* indicates successful termination */
}

The string "1000000" converted to long int is 1000000
The converted value divided by 2 is 500000
*/ Fig. 8.9: fig08_09.c
    Using strtod */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    /* initialize string pointer */
    const char *string = "51.2% are admitted";
    double d;    /* variable to hold converted sequence */
    char *stringPtr; /* create char pointer */
    d = strtod( string, &stringPtr );

    printf( "The string "%s" is converted to the\n", string );
    printf( "double value %2f and the string "%s"\n", d, stringPtr );
    return 0; /* indicates successful termination */
} /* end main */

The string "51.2% are admitted" is converted to the
double value 51.20 and the string "% are admitted"
/* Fig. 8.10: fig08_10.c  
Using strtol */

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

int main()
{
    const char *string = "-1234567abc"; /* initialize string pointer */

    char *remainderPtr; /* create char pointer */
    long x;             /* variable to hold converted sequence */

    x = strtol( string, &remainderPtr, 0 );

    printf( "The original string is ", string,
    "The converted value is ", x,
    "The remainder of the original string is ",
    remainderPtr,
    "The converted value plus 567 is ", x + 567 );

    return 0; /* indicates successful termination */
} /* end main */

The original string is "-1234567abc"
The converted value is -1234567
The remainder of the original string is "abc"
The converted value plus 567 is -1234000
/* Fig. 8.11: fig08_11.c
Using strtoul */

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

int main()
{
    const char *string = "1234567abc"; /* initialize string pointer */
    unsigned long x;    /* variable to hold converted sequence */
    char *remainderPtr; /* create char pointer */

    x = strtoul( string, &remainderPtr, 0 );

    printf( "The original string is ", string,
            "The converted value is ", x,
            "The remainder of the original string is ",
            remainderPtr,
            "The converted value minus 567 is ", x - 567 );

    return 0; /* indicates successful termination */
}

The original string is "1234567abc"
The converted value is 1234567
The remainder of the original string is "abc"
The converted value minus 567 is 1234000
Functions in `<stdio.h>`

- **Used to manipulate character and string data**

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int getchar( void );</code></td>
<td>Inputs the next character from the standard input and returns it as an integer.</td>
</tr>
<tr>
<td><code>char *gets( char *s );</code></td>
<td>Inputs characters from the standard input into the array <code>s</code> until a newline or end-of-file character is encountered. A terminating null character is appended to the array.</td>
</tr>
<tr>
<td><code>int putchar( int c );</code></td>
<td>Prints the character stored in <code>c</code>.</td>
</tr>
<tr>
<td><code>int puts( const char *s );</code></td>
<td>Prints the string <code>s</code> followed by a newline character.</td>
</tr>
<tr>
<td><code>int sprintf( char *s, const char *format, ... );</code></td>
<td>Equivalent to <code>printf</code>, except the output is stored in the array <code>s</code> instead of printing it on the screen.</td>
</tr>
<tr>
<td><code>int sscanf( char *s, const char *format, ... );</code></td>
<td>Equivalent to <code>scanf</code>, except the input is read from the array <code>s</code> instead of reading it from the keyboard.</td>
</tr>
</tbody>
</table>
/* Fig. 8.13: fig08_13.c */

Using gets and putchar */

#include <stdio.h>

int main()
{
    char sentence[80]; /* create char array */

    void reverse( const char * const sPtr ); /* prototype */

    printf( "Enter a line of text:\n" );

    /* use gets to read line of text */
    gets( sentence );

    printf( "\nThe line printed backwards is:\n" );
    reverse( sentence );

    return 0; /* indicates successful termination */
}

/* end main */
Enter a line of text:
Characters and Strings

The line printed backwards is:
sgnirtS dna sretcarahC

Enter a line of text:
able was I ere I saw elba

The line printed backwards is:
able was I ere I saw elba

/* recursively outputs characters in string in reverse order */

void reverse( const char * const sPtr )
{
    /* if end of the string */
    if ( sPtr[ 0 ] == '\0' ) {
        return;
    } /* end if */
    else { /* if not end of the string */
        reverse( &sPtr[ 1 ] );
        putchar( sPtr[ 0 ] ); /* use putchar to display character */
    } /* end else */
} /* end function reverse */
/* Fig. 8.14: fig08_14.c  
Using getchar and puts */
#include <stdio.h>

int main()
{
    char c;          /* variable to hold character input by user */
    char sentence[80]; /* create char array */
    int i = 0;        /* initialize counter i */

    /* prompt user to enter line of text */
    puts("Enter a line of text:");

    /* use getchar to read each character */
    while ( ( c = getchar() ) != '\n' ) {
        sentence[ i++ ] = c;
    } /* end while */

    sentence[ i ] = '\0';

    /* use puts to display sentence */
    puts("The line entered was:");
    puts( sentence );

    return 0; /* indicates successful termination */
} /* end main */
String Manipulation Functions of the String Handling Library

- String handling library has functions to
  - Manipulate string data
  - Search strings
  - Tokenize strings
  - Determine string length

### Function Prototype and Description

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char *strcpy( char *s1, const char *s2 )</code></td>
<td>Copies string ( s_2 ) into array ( s_1 ). The value of ( s_1 ) is returned.</td>
</tr>
<tr>
<td><code>char *strncpy( char *s1, const char *s2, size_t n )</code></td>
<td>Copies at most ( n ) characters of string ( s_2 ) into array ( s_1 ). The value of ( s_1 ) is returned.</td>
</tr>
<tr>
<td><code>char *strcat( char *s1, const char *s2 )</code></td>
<td>Appends string ( s_2 ) to array ( s_1 ). The first character of ( s_2 ) overwrites the terminating null character of ( s_1 ). The value of ( s_1 ) is returned.</td>
</tr>
<tr>
<td><code>char *strncat( char *s1, const char *s2, size_t n )</code></td>
<td>Appends at most ( n ) characters of string ( s_2 ) to array ( s_1 ). The first character of ( s_2 ) overwrites the terminating null character of ( s_1 ). The value of ( s_1 ) is returned.</td>
</tr>
</tbody>
</table>
/* Fig. 8.18: fig08_18.c */

using strcpy and strncpy */

#include <stdio.h>

#include <string.h>

int main()
{
    char x[] = "Happy Birthday to You"; /* initialize char array x */
    char y[25];                       /* create char array y */
    char z[15];                       /* create char array z */

    /* copy contents of x into y */
    printf("The string in array x is: ", x,
           "The string in array y is: ", strcpy(y, x));

    /* copy first 14 characters of x into z. Does not copy null character */
    strncpy(z, x, 14);

    z[14] = '\0'; /* append '\0' to z's contents */
    printf("The string in array z is: %s", z);

    return 0; /* indicates successful termination */
}

The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
/* Fig. 8.19: fig08_19.c
   Using strcat and strncat */
#include <stdio.h>
#include <string.h>

int main()
{
    char s1[20] = "Happy "; /* initialize char array s1 */
    char s2[] = "New Year "; /* initialize char array s2 */
    char s3[40] = "";       /* initialize char array s3 */

    printf( "s1 = %s\n s2 = %s\n", s1, s2 );

    /* concatenate s2 to s1 */
    printf( "strcat( s1, s2 ) = %s\n", strcat( s1, s2 ) );

    /* concatenate first 6 characters of s1 to s3. Place '\0'
     after last character */
    printf( "strncat( s3, s1, 6 ) = %s\n", strncat( s3, s1, 6 ) );

    /* concatenate s1 to s3 */
    printf( "strcat( s3, s1 ) = %s\n", strcat( s3, s1 ) );

    return 0; /* indicates successful termination */
} /* end main */
Comparing strings

- Computer compares numeric ASCII codes of characters in string
- Appendix D has a list of character codes

```c
int strcmp( const char *s1, const char *s2 );
```

- Compares string `s1` to `s2`
- Returns a negative number if `s1 < s2`, zero if `s1 == s2` or a positive number if `s1 > s2`

```c
int strncmp( const char *s1, const char *s2, size_t n );
```

- Compares up to `n` characters of string `s1` to `s2`
- Returns values as above
/ * Fig. 8.21: fig08_21.c *
  Using strcmp and strncmp */

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

int main()
{
  const char *s1 = "Happy New Year"; /* initialize char pointer */
  const char *s2 = "Happy New Year"; /* initialize char pointer */
  const char *s3 = "Happy Holidays"; /* initialize char pointer */

  printf("%s%s
%s%s
%s%s

%s%2d
%s%2d
%s%2d

",  
    "s1 = ", s1, "s2 = ", s2, "s3 = ", s3,
    "strcmp(s1, s2) = ", strcmp( s1, s2 ),
    "strcmp(s1, s3) = ", strcmp( s1, s3 ),
    "strcmp(s3, s1) = ", strcmp( s3, s1 )
    );

  printf(" \n",  
    "strncmp(s1, s3, 6) = ", strncmp( s1, s3, 6 ),
    "strncmp(s1, s3, 7) = ", strncmp( s1, s3, 7 ),
    "strncmp(s3, s1, 7) = ", strncmp( s3, s1, 7 )
    );

  return 0; /* indicates successful termination */

} /* end main */
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

\text{strcmp}(s1, s2) = 0
\text{strcmp}(s1, s3) = 1
\text{strcmp}(s3, s1) = -1

\text{strncmp}(s1, s3, 6) = 0
\text{strncmp}(s1, s3, 7) = 1
\text{strncmp}(s3, s1, 7) = -1
<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char *strchr( const char *s, int c );</code></td>
<td>Locates the first occurrence of character <code>c</code> in string <code>s</code>. If <code>c</code> is found, a pointer to <code>c</code> in <code>s</code> is returned. Otherwise, a <code>NULL</code> pointer is returned.</td>
</tr>
<tr>
<td><code>size_t strcspn( const char *s1, const char *s2 );</code></td>
<td>Determines and returns the length of the initial segment of string <code>s1</code> consisting of characters not contained in string <code>s2</code>.</td>
</tr>
<tr>
<td><code>size_t strspn( const char *s1, const char *s2 );</code></td>
<td>Determines and returns the length of the initial segment of string <code>s1</code> consisting only of characters contained in string <code>s2</code>.</td>
</tr>
<tr>
<td><code>char *strpbrk( const char *s1, const char *s2 );</code></td>
<td>Locates the first occurrence in string <code>s1</code> of any character in string <code>s2</code>. If a character from string <code>s2</code> is found, a pointer to the character in string <code>s1</code> is returned. Otherwise, a <code>NULL</code> pointer is returned.</td>
</tr>
<tr>
<td><code>char *strstr( const char *s1, const char *s2 );</code></td>
<td>Locates the first occurrence in string <code>s1</code> of string <code>s2</code>. If the string is found, a pointer to the string in <code>s1</code> is returned. Otherwise, a <code>NULL</code> pointer is returned.</td>
</tr>
<tr>
<td><code>char *strtok( char *s1, const char *s2 );</code></td>
<td>A sequence of calls to <code>strtok</code> breaks string <code>s1</code> into &quot;tokens&quot;—logical pieces such as words in a line of text—separated by characters contained in string <code>s2</code>. The first call contains <code>s1</code> as the first argument, and subsequent calls to continue tokenizing the same string contain <code>NULL</code> as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, <code>NULL</code> is returned.</td>
</tr>
</tbody>
</table>
/* Fig. 8.23: fig08_23.c */

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

int main()
{
    const char *string = "This is a test"; /* initialize char pointer */
    char character1 = 'a'; /* initialize character1 */
    char character2 = 'z'; /* initialize character2 */

    /* if character1 was found in string */
    if ( strchr( string, character1 ) != NULL ) {
        printf( "\'%c\' was found in "string",
                character1, string );
    } /* end if */
    else { /* if character1 was not found */
        printf( "\'%c\' was not found in "string",
                character1, string );
    } /* end else */
} /* end main() */
/* if character2 was found in string */
if ( strchr( string, character2 ) != NULL ) {
    printf( "'\%c' was found in "string",
            character2, string );
} /* end if */

else { /* if character2 was not found */
    printf( "'\%c' was not found in "string",
            character2, string );
} /* end else */

return 0; /* indicates successful termination */

/* end main */
/* Fig. 8.24: fig08_24.c */

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

int main()
{
    /* initialize two char pointers */
    const char *string1 = "The value is 3.14159";
    const char *string2 = "1234567890";

    printf( "%s%s
%s%s

%s
%s%u",
        "string1 = ", string1, "string2 = ", string2,
        "The length of the initial segment of string1",
        "containing no characters from string2 = ",
        strcspn( string1, string2 ) );

    return 0; /* indicates successful termination */
}

The length of the initial segment of string1
containing no characters from string2 = 13
/* Fig. 8.25: fig08_25.c
Using strpbrk */
#include <stdio.h>
#include <string.h>

int main()
{
    const char *string1 = "This is a test"; /* initialize char pointer */
    const char *string2 = "beware";       /* initialize char pointer */

    printf( "%s"%s"%c"%s
"%s"
"%s"
", string1, string2, *strpbrk( string1, string2 ),
            " is the first character to appear in ", string1 );

    return 0; /* indicates successful termination */
}

Of the characters in "beware"
'a' is the first character to appear in
"This is a test"
/* Fig. 8.26: fig08_26.c
   Using strstr */
#include <stdio.h>
#include <string.h>

int main()
{
    /* initialize char pointer */
    const char *string1 = "A zoo has many animals including zebras"
    int c = 'z'; /* initialize c */

    printf( "The remainder of string1 beginning with the last occurrence of character 'z' is: "
            , strstr( string1, c )
    );

    return 0; /* indicates successful termination */
}

The remainder of string1 beginning with the last occurrence of character 'z' is: "zebras"
/* Fig. 8.27: fig08_27.c
Using strspn */

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

int main()
{
    /* initialize two char pointers */
    const char *string1 = "The value is 3.14159";
    const char *string2 = "aehi lsTuv";

    printf("%s%sn%sn%sn%sn%sn%sn%sn%sn%sn\n","string1 = ", string1, "string2 = ", string2,
          "The length of the initial segment of string1",
          "containing only characters from string2 = ",
          strspn( string1, string2 )");

    return 0; /* indicates successful termination */
}

string1 = The value is 3.14159
string2 = aehi lsTuv

The length of the initial segment of string1
containing only characters from string2 = 13
/* Fig. 8.28: fig08_28.c  
   Using strstr */
#include <stdio.h>
#include <string.h>

int main()
{
    const char *string1 = "abcdefabcdef"; /* initialize char pointer */
    const char *string2 = "def";          /* initialize char pointer */

    printf( "%s%s
%s%s

%s
%s%s
",  
        "string1 = ", string1, "string2 = ", string2,  
        "The remainder of string1 beginning with the",  
        "first occurrence of string2 is: ",  
        strstr( string1, string2 ) );

    return 0; /* indicates successful termination */
}

string1 = abcdefabcdef
string2 = def

The remainder of string1 beginning with the 
first occurrence of string2 is: defabcdef
/Fig. 8.29: fig08_29.c

#include <stdio.h>

#include <string.h>

int main()
{
    /* initialize array string */
    char string[] = "This is a sentence with 7 tokens";
    char *tokenPtr; /* create char pointer */

    printf("%s
%s

%s
", 
        "The string to be tokenized is: ", string, 
        "The tokens are: ");

    tokenPtr = strtok( string, " "); /* begin tokenizing sentence */

    /* continue tokenizing sentence until tokenPtr becomes NULL */
    while ( tokenPtr != NULL )
    {
        printf("%s
", tokenPtr);
        printf("%s
", tokenPtr);
        tokenPtr = strtok( NULL, " "); /* get next token */
    } /* end while */

    return 0; /* indicates successful termination */
} /* end main */
Memory Functions of the String-handling Library

Memory Functions
- In `<stdlib.h>`
- Manipulate, compare, and search blocks of memory
- Can manipulate any block of data

**Pointer parameters are `void *`**
- Any pointer can be assigned to `void *`, and vice versa
- `void *` cannot be dereferenced
  - Each function receives a size argument specifying the number of bytes (characters) to process
### Memory Functions of the String-handling Library

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void *memcpy( void *s1, const void *s2, size_t n );</code></td>
<td>Copies ( n ) characters from the object pointed to by ( s2 ) into the object pointed to by ( s1 ). A pointer to the resulting object is returned.</td>
</tr>
<tr>
<td><code>void *memmove( void *s1, const void *s2, size_t n );</code></td>
<td>Copies ( n ) characters from the object pointed to by ( s2 ) into the object pointed to by ( s1 ). The copy is performed as if the characters were first copied from the object pointed to by ( s2 ) into a temporary array and then from the temporary array into the object pointed to by ( s1 ). A pointer to the resulting object is returned.</td>
</tr>
<tr>
<td><code>int memcmp( const void *s1, const void *s2, size_t n );</code></td>
<td>Compares the first ( n ) characters of the objects pointed to by ( s1 ) and ( s2 ). The function returns 0, less than 0 or greater than 0 if ( s1 ) is equal to, less than or greater than ( s2 ).</td>
</tr>
<tr>
<td><code>void *memchr( const void *s, int c, size_t n );</code></td>
<td>Locates the first occurrence of ( c ) (converted to unsigned char) in the first ( n ) characters of the object pointed to by ( s ). If ( c ) is found, a pointer to ( c ) in the object is returned. Otherwise, NULL is returned.</td>
</tr>
<tr>
<td><code>void *memset( void *s, int c, size_t n );</code></td>
<td>Copies ( c ) (converted to unsigned char) into the first ( n ) characters of the object pointed to by ( s ). A pointer to the result is returned.</td>
</tr>
</tbody>
</table>
/* Fig. 8.31: fig08_31.c */

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

int main()
{
    char s1[17];  /* create char array s1 */
    char s2[] = "Copy this string";  /* initialize char array s2 */

    memcpy(s1, s2, 17);
    printf("After s2 is copied into s1 with memcpy, ",
           "s1 contains ", s1);

    return 0; /* indicates successful termination */
}

After s2 is copied into s1 with memcpy,
s1 contains "Copy this string"
/* Fig. 8.32: fig08_32.c */

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

int main()
{
    char x[] = "Home Sweet Home"; /* initialize char array x */

    printf( "The string in array x before memmove is: ", x );
    printf( "The string in array x after memmove is: ",
            memmove( x, &x[ 5 ], 10 ) );

    return 0; /* indicates successful termination */
}

The string in array x before memmove is: Home Sweet Home
The string in array x after memmove is: Sweet Home Home
/ * Fig. 8.33: fig08_33.c
   Using memcmp */
#include <stdio.h>
#include <string.h>

int main()
{
    char s1[] = "ABCDEFG"; /* initialize char array s1 */
    char s2[] = "ABCDXYZ"; /* initialize char array s2 */

    printf( "%s%s
%s%s

%s%2d
%s%2d
%s%2d
",
            "s1 = " , s1 , "s2 = " , s2,
            "memcmp( s1, s2, 4 ) = " , memcmp( s1, s2, 4 ) ,
            "memcmp( s1, s2, 7 ) = " , memcmp( s1, s2, 7 ) ,
            "memcmp( s2, s1, 7 ) = " , memcmp( s2, s1, 7 ) );

    return 0; /* indicate successful termination */
}

s1 = ABCDEFG
s2 = ABCDXYZ

memcmp( s1, s2, 4 ) = 0
memcmp( s1, s2, 7 ) = -1
memcmp( s2, s1, 7 ) = 1
/* Fig. 8.34: fig08_34.c */
#include <stdio.h>
#include <string.h>

int main()
{
    const char *s = "This is a string"; /* initialize char pointer */

    printf( "%s'\%c'\%s"%s"
", "The remainder of s after character ", 'r', " is found is ", memchr( s, 'r', 16 ) );

    return 0; /* indicates successful termination */
} /* end main */

The remainder of s after character 'r' is found is "ring"
Other Functions of the String Handling Library

- `char *strerror( int errornum );`
  - Creates a system-dependent error message based on `errornum`
  - Returns a pointer to the string

- `size_t strlen( const char *s );`
  - Returns the number of characters (before `NULL`) in string `s`
/* Fig. 8.35: fig08_35.c
Using memset */
#include <stdio.h>
#include <string.h>

int main()
{
    char string1[15] = "BBBBBBBBBBBBB"; /* initialize string1 */
    printf( "string1 = %s\n", string1 );
    printf( "string1 after memset = %s\n", memset( string1, 'b', 7 ) );
    return 0; /* indicates successful termination */
}

string1 = BBBBBBBBBBBBBBB
string1 after memset = bbbbbbbBBBBBBB
/* Fig. 8.37: fig08_37.c */

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

int main()
{
    printf( "\n", strerror( 2 ) );

    return 0; /* indicates successful termination */
}

/* end main */

No such file or directory
/* Fig. 8.38: fig08_38.c
   Using strlen */
#include <stdio.h>
#include <string.h>

int main()
{
    /* initialize 3 char pointers */
    const char *string1 = "abcdefghijklmnopqrstuvwxyz";
    const char *string2 = "four";
    const char *string3 = "Boston";

    printf("The length of ", string1, " is ",
           ( unsigned long ) strlen( string1 ),
           "The length of ", string2, " is ",
           ( unsigned long ) strlen( string2 ),
           "The length of ", string3, " is ",
           ( unsigned long ) strlen( string3 ));

    return 0; /* indicates successful termination */
}

The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
Structures

- Collections of related variables (aggregates) under one name
  - Can contain variables of different data types
- Commonly used to define records to be stored in files
- Combined with pointers, can create linked lists, stacks, queues, and trees
Example

```c
struct card {
    char *face;
    char *suit;
};
```

- `struct` introduces the definition for structure `card`.
- `card` is the structure name and is used to declare variables of the structure type.
- `card` contains two members of type `char *`
  - These members are `face` and `suit`.
struct information

- A struct cannot contain an instance of itself
- Can contain a member that is a pointer to the same structure type
- A structure definition does not reserve space in memory
  - Instead creates a new data type used to define structure variables

Definitions

- Defined like other variables:
  ```
  card oneCard, deck[52], *cPtr;
  ```

- Can use a comma separated list:
  ```
  struct card {
      char *face;
      char *suit;
  } oneCard, deck[52], *cPtr;
  ```
### Structure Definitions

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>01100001</strong></td>
<td><strong>00000000</strong></td>
<td><strong>01100001</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 10.1** A possible storage alignment for a variable of type `struct example` showing an undefined area in memory. §
**Valid Operations**

- Assigning a structure to a structure of the same type
- Taking the address (&) of a structure
- Accessing the members of a structure
- Using the `sizeof` operator to determine the size of a structure
Initializing Structures

- **Initializer lists**
  - Example:
    ```java
card oneCard = { "Three", "Hearts" };
```

- **Assignment statements**
  - Example:
    ```java
card threeHearts = oneCard;
```
  - Could also define and initialize `threeHearts` as follows:
    ```java
card threeHearts;
    threeHearts.face = "Three";
    threeHearts.suit = "Hearts";
```
Accessing Members of Structures

- **Accessing structure members**
  - Dot operator (.) used with structure variables
    ```c
    card myCard;
    printf( "%s", myCard.suit );
    ```
  - Arrow operator (->) used with pointers to structure variables
    ```c
    card *myCardPtr = &myCard;
    printf( "%s", myCardPtr->suit );
    ```
    - `myCardPtr->suit` is equivalent to
      ```c
      (*myCardPtr).suit
      ```
/* Fig. 10.2: fig10_02.c

Using the structure member and
structure pointer operators */

#include <stdio.h>

/* card structure definition */
struct card {
    char *face; /* define pointer face */
    char *suit; /* define pointer suit */
}; /* end structure card */

int main()
{
    struct card a;    /* define struct a */
    struct card *aPtr; /* define a pointer to card */

    /* place strings into card structures */
a.face = "Ace";
a.suit = "Spades";

    aPtr = &a; /* assign address of a to aPtr */
23  printf("%s\n%sn%sn%sn\n", a.face, " of ", a.suit, 
    aPtr->face, " of ", aPtr->suit, 
    ( *aPtr ).face, " of ", ( *aPtr ).suit );
24
25  return 0; /* indicates successful termination */
26
27} /* end main */
Using Structures With Functions

- **Passing structures to functions**
  - Pass entire structure
    - Or, pass individual members
  - Both pass call by value

- **To pass structures call-by-reference**
  - Pass its address
  - Pass reference to it

- **To pass arrays call-by-value**
  - Create a structure with the array as a member
  - Pass the structure
typedef

- Creates synonyms (aliases) for previously defined data types
- Use `typedef` to create shorter type names
- Example:
  ```c
  typedef struct Card *CardPtr;
  ```
- Defines a new type name `CardPtr` as a synonym for `struct Card *`
- `typedef` does not create a new data type
  - Only creates an alias
Example: High-Performance Card-shuffling and Dealing Simulation

- **Pseudocode:**
  - Create an array of card structures
  - Put cards in the deck
  - Shuffle the deck
  - Deal the cards
/* Fig. 10.3: fig10_03.c */

The card shuffling and dealing program using structures */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* card structure definition */
struct card {
    const char *face; /* define pointer face */
    const char *suit; /* define pointer suit */
}; /* end structure card */

typedef struct card Card;

/* prototypes */
void fillDeck( Card * const wDeck, const char * wFace[],
               const char * wSuit[] );
void shuffle( Card * const wDeck );
void deal( const Card * const wDeck );

int main()
{
    Card deck[ 52 ]; /* define array of Cards */
    /* code here */
}
/ * initialize array of pointers */
const char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
    "Six", "Seven", "Eight", "Nine", "Ten",
    "Jack", "Queen", "King"};

/* initialize array of pointers */
const char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades"};

srand( time( NULL ) ); /* randomize */

fillDeck( deck, face, suit ); /* load the deck with Cards */
shuffle( deck ); /* put Cards in random order */
deal( deck ); /* deal all 52 Cards */

return 0; /* indicates successful termination */

} /* end main */

/* place strings into Card structures */
void fillDeck( Card *const wDeck, const char *wFace[],
    const char *wSuit[] )
{
    int i; /* counter */
/* loop through wDeck */
for ( i = 0; i <= 51; i++ ) {
    wDeck[ i ].face = wFace[ i % 13 ];
    wDeck[ i ].suit = wSuit[ i / 13 ];
} /* end for */

} /* end function fillDeck */

/* shuffle cards */
void shuffle( Card * const wDeck )
{
    int i;     /* counter */
    int j;     /* variable to hold random value between 0 - 51 */
    Card temp; /* define temporary structure for swapping Cards */

    /* loop through wDeck randomly swapping Cards */
    for ( i = 0; i <= 51; i++ ) {
        j = rand() % 52;
        temp = wDeck[ i ];
        wDeck[ i ] = wDeck[ j ];
        wDeck[ j ] = temp;
    } /* end for */

} /* end function shuffle */
74 /* deal cards */
75 void deal( const Card * const wDeck )
76 {
77   int i; /* counter */
78
79   /* loop through wDeck */
80   for ( i = 0; i <= 51; i++ ) {
81     printf( "%s of %s%c" , wDeck[ i ].face, wDeck[ i ].suit,
82             ( i + 1 ) % 2 ? 't' : '\n' );
83   } /* end for */
84
85 } /* end function deal */