Introduction to Structured Programming

1

Text Book & Reference Books

- Text Book:
  - P.J. Deitel and H.M. Deitel, "C How to Program", Fifth Edition, Pearson Prentice-Hall, 2007.
- Reference Books:
  - Jeri R. Hanly and Elliot B. Koffman, "Problem Solving and Program Design in C", Fifth Edition, Addison Wesley, 2007.
  - Brian W. Kernighan and Dennis M. Ritchie, "Programming Language", Second Edition, Prentice Hall, PTR, 1988.
  - Alice E. Fischer, David W. Eggert and Stephen M. Ross, "Applied C: An Introduction and More", McGraw Hill, 2001.
  - Anany Levitin, "Introduction to The Design and Analysis of Algorithms", Pearson Addison Wesley, 2003.

2

Course Outline

- Programming Methodologies
  - Algorithm Design
  - Flowcharts and Pseudo Codes
  - Top-down Program Design
- ★ Structured Language (focus on **C Language**)
  - Data Types, Declarations, Operators and Expressions
  - Input and Output
  - Control Structures
  - Functions and Program Structure
  - Pointers
  - Arrays, Strings, and Unions
  - File Processing
- Discipline of Programming
  - Programming Style
  - Structured Coding and Program Modularity
  - Program Documentation and Maintenance

3

Chapter 1

Introduction to Computers, the Internet and the Web

Acknowledgment

The notes are adapted from those provided by Deitel & Associates, Inc. and Pearson Education Inc.

4

OBJECTIVES

- Basic computer concepts
- Different types of programming languages
- C programming language
  - History
  - C Standard Library
  - Object-oriented Programming
  - Typical C Program Development Environment
- The history of the Internet and the World Wide Web
- Review

5

What is a Computer?

- Computer
  - Device capable of performing computations and making logical decisions (hardware)
  - Computers process data under the control of sets of instructions called computer programs (software)
- Hardware
  - Various devices comprising a computer, such as central processing unit (CPU), memory, motherboard and hard disks as well as peripheral devices (keyboard, screen, mouse CD-ROM)
  - Hardware trends – Every year or two the following approximately double:
    - Amount of memory in which to execute programs
    - The processor speed at which computers execute their programs
- Software
  - Programs that run on a computer

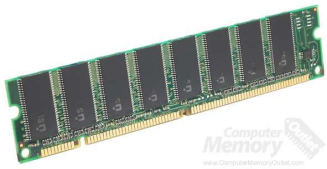
6

Computer Organization

- Six logical units in every computer:
  - Input unit
    - Obtains information from input devices (keyboard, mouse)
  - Output unit
    - Outputs information (to screen, to printer, to control other devices)
  - Memory unit
    - Rapid access, low capacity, stores input information
  - Arithmetic and logic unit (ALU)
    - Performs arithmetic calculations and logic decisions
  - Central processing unit (CPU)
    - Supervises and coordinates the other sections of the computer
    - ALU is now a fundamental building block of the CPU
  - Secondary storage unit
    - Cheap, long-term, high-capacity storage
    - Stores inactive programs

7

Hardware



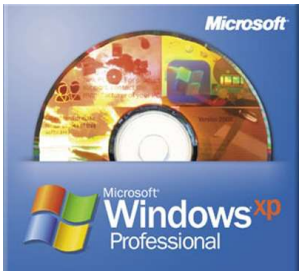
8

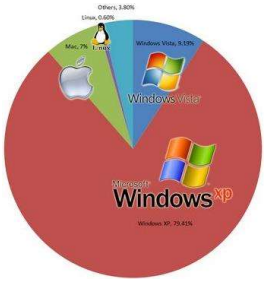
Evolution of Operating Systems

- Batch processing
  - Do only one job or task at a time
- Operating systems
  - Manage transitions between jobs
  - Increased throughput
    - Amount of work computers process
- Multiprogramming
  - Computer resources are shared by many jobs or tasks
- Timesharing
  - Computer runs a small portion of one user's job then moves on to service the next user

9

Operating Systems





Macintosh

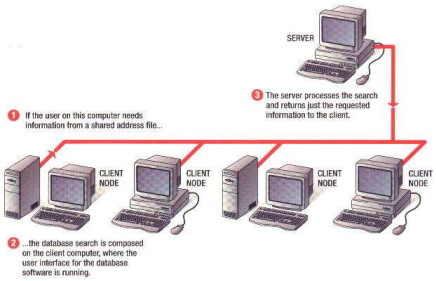
10

Personal Computing, Distributed Computing, and Client/Server Computing

- Personal computers
  - Economical enough for individual
- Distributed computing
  - Computing distributed over networks
- Client/server computing
  - Sharing of information across computer networks between file servers and clients (personal computers)

11

Personal Computing, Distributed Computing, and Client/Server Computing



12

OBJECTIVES

- Basic computer concepts
- Different types of programming languages
- C programming language
  - History
  - C Standard Library
  - Object-oriented Programming
  - Typical C Program Development Environment
- The history of the Internet and the World Wide Web
- Review

13

Machine Languages, Assembly Languages, and High-level Languages

Three types of programming languages

1. Machine languages
   - Strings of numbers giving machine specific instructions
   - Example: `+1300042774`
2. Assembly languages
   - English-like abbreviations representing elementary computer operations (translated via assemblers)
   - Example: `LOAD BASEPAY`
3. High-level languages
   - Codes similar to everyday English
   - Use mathematical notations (translated via compilers)
   - Example: `grossPay = basePay + overTimePay`

14

Fortran, COBOL, Pascal and Ada

- FORTRAN
  - Developed by IBM Corporation in the 1950s
  - Used for scientific and engineering applications that require complex mathematical computations
- COBOL
  - Developed in 1959 by computer manufacturers, the government and industrial computer users
  - Used for commercial applications that require precise and efficient manipulation of large amounts of data Pascal
- Pascal
  - Developed by Professor Niklaus Wirth in 1971
  - Designed for teaching structured programming
- Ada
  - Developed under the sponsorship of the U.S. Department of Defense (DOD) during the 1970s and early 1980s
  - Able to perform multitasking

15

OBJECTIVES

- Basic computer concepts
- Different types of programming languages
- C programming language
  - History
  - C Standard Library
  - Object-oriented Programming
  - Typical C Program Development Environment
- The history of the Internet and the World Wide Web
- Review

16

History of C

- C
  - Evolved by Ritchie from two previous programming languages, BCPL and B
  - Used to develop UNIX
  - Used to write modern operating systems
  - Hardware independent (portable)
  - By late 1970's C had evolved to “Traditional C”
- Standardization
  - Many slight variations of C existed, and were incompatible
  - Committee formed to create a "unambiguous, machine-independent" definition
  - Standard created in 1989, updated in 1999

17

C Standard Library

- C programs consist of pieces/modules called functions
  - A programmer can create his own functions
    - Advantage: the programmer knows exactly how it works
    - Disadvantage: time consuming
  - Programmers will often use the C library functions
    - Use these as building blocks
  - Avoid re-inventing the wheel
    - If a premade function exists, generally best to use it rather than write your own
    - Library functions carefully written, efficient, and portable

18

OBJECTIVES

- Basic computer concepts
- Different types of programming languages
- C programming language
  - History
  - C Standard Library
  - Object-oriented Programming
  - Typical C Program Development Environment
- The history of the Internet and the World Wide Web
- Review

19

The Key Software Trend: Object Technology

- Objects
  - Reusable software components that model items in the real world
  - Meaningful software units
    - Date objects, time objects, paycheck objects, invoice objects, audio objects, video objects, file objects, record objects, etc.
    - Any noun can be represented as an object
  - Very reusable
  - More understandable, better organized, and easier to maintain than procedural programming
  - Favor modularity

20

C++

- C++
  - Superset of C developed by Bjarne Stroustrup at Bell Labs
  - "Spruces up" C, and provides object-oriented capabilities
  - Object-oriented design very powerful
    - 10 to 100 fold increase in productivity
  - Dominant language in industry and academia
- Learning C++
  - Because C++ includes C, some feel it is best to master C, then learn C++
  - More details of C++, see Chapter 18-27 of the textbook

21

Java

- Java is used to
  - Create Web pages with dynamic and interactive content
  - Develop large-scale enterprise applications
  - Enhance the functionality of Web servers
  - Provide applications for consumer devices (such as cell phones, pagers and personal digital assistants)
- Java How to Program
  - Closely followed the development of Java by Sun
  - Teaches first-year programming students the essentials of graphics, images, animation, audio, video, database, networking, multithreading and collaborative computing

22

BASIC, Visual Basic, Visual C++, Visual C# and .NET

- BASIC
  - Developed in the mid-1960s by Professors John Kemeny and Thomas Kurtz of Dartmouth College as a language for writing simple programs
- Visual Basic
  - Introduced by Microsoft in 1991 to simplify the process of making Windows applications
- Visual Basic, Visual C++, and Visual C#
  - Designed for Microsoft's .NET programming platform

23

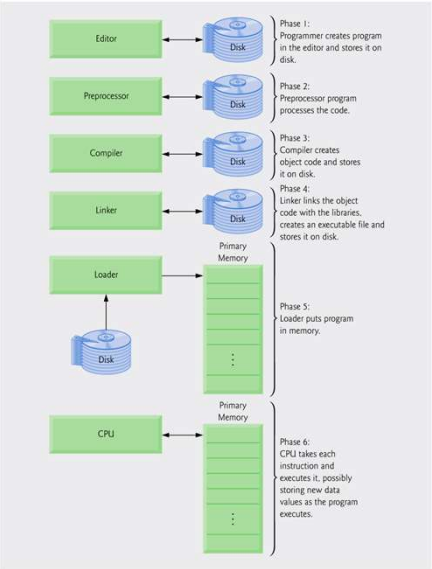
OBJECTIVES

- Basic computer concepts
- Different types of programming languages
- C programming language
  - History
  - C Standard Library
  - Object-oriented Programming
  - Typical C Program Development Environment
- The history of the Internet and the World Wide Web
- Review

24

A Typical C Program Development Environment

- Phases of C Programs:
  - Edit (C program file names should end with the .c extension)
  - Preprocess
  - Compile
  - Link
  - Load
  - Execute



25

A Typical C Program Development Environment

- Phase 1: Create your program with an editor program
- Phase 2: A preprocessor find some preprocessor directives (#include <stdio.h>) to include other files and perform various text replacements
- Phase 3: Compiler compiles the program into machine languages (object code != object-oriented programming).
- Phase 4: A linker links the object code with the code in library or other places to produce an executable image.
- Phase 5: A loader loads the executable image into memory (RAM).
- Phase 6: CPU executes the program one instruction at a time. The load process in Windows OS is just input the name of the executable file (for example, lab01.exe).

26

OBJECTIVES

- Basic computer concepts
- Different types of programming languages
- C programming language
  - History
  - C Standard Library
  - Object-oriented Programming
  - Typical C Program Development Environment
- The history of the Internet and the World Wide Web
- Review

27

History of the Internet

- The Internet enables
  - Quick and easy communication via e-mail
  - International networking of computers
- Packet switching
  - The transfer of digital data via small packets
  - Allows multiple users to send and receive data simultaneously
- No centralized control
  - If one part of the Internet fails, other parts can still operate
- TCP/IP
- Bandwidth
  - Information carrying capacity of communications lines

28

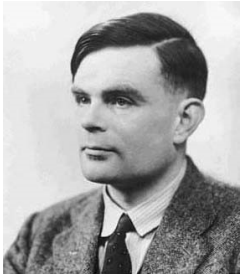
History of the World Wide Web (WWW)

- World Wide Web
  - Locate and view multimedia-based documents on almost any subject
  - Makes information instantly and conveniently accessible worldwide
  - Possible for individuals and small businesses to get worldwide exposure
  - Changing the way business is done

29

VIP in Computing

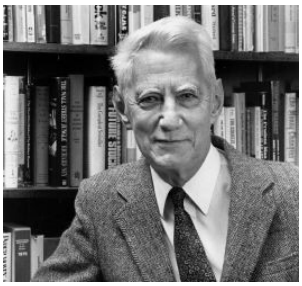
- **Alan Turing**: a British mathematician who is the father of theoretical computer science and artificial intelligence.
  - The **Turing Award** is given annually by the Association for Computing Machinery (ACM) to "an individual selected for contributions of a technical nature made to the computing community. The contributions should be of lasting and major technical importance to the computer field". Often recognized as the "Nobel Prize of computing". As of 2007, the award is accompanied by a prize of $250,000, co-sponsored by Intel and Google.



30

VIP in Communication

- **Claude E. Shannon** is famous for having founded information theory with one landmark paper published in 1948 "A Mathematical Theory of Communication".



31

OBJECTIVES

- Basic computer concepts
- Different types of programming languages
- C programming language
  - History
  - C Standard Library
  - Object-oriented Programming
  - Typical C Program Development Environment
- The history of the Internet and the World Wide Web
- Review

32

Computer Organization

- ★ Hardware: Input/Output; **RAM** (Random-access memory); Hard disk (secondary storage unit); ALU/**CPU**; Motherboard; Computer Case (power); CD or DVD-ROM (peripheral components).
- Moore' Law: the same price for the double of computing power approximately every one or two years
- Software: Operating systems (**OS**); Application Software. (A hierarchical structure)
- ★ OS manages and coordinates activities and the sharing of the limited resources of the computer. Applications use **application programming interface** (**API**) provided by libraries and OS to handle computer resources.

33

Languages

- ★ Machine languages (a numerical representation `+1300042774`) are **hardware or platform dependent**. They are understood directly by ALU/CPU and thus **efficient** to computers. Every CPU model has its own machine code, or instruction sets.
- Assembly languages (a symbolic representation `LOAD BASEPAY`) are also architecture dependent. It is difficult to portable to other systems or machines. Today, it is used primarily for direct hardware manipulation. Typical uses are device drivers, low-level embedded systems, and real-time systems.
- ★ High-level languages like C are more **potable** to other platforms and easy to understand by programmers in an everyday English format (`grossPay = basePay + overTimePay`).

34

C

- **C standard library** contains of modules called **functions (building-block approach)**. We do not need to re-invent the wheel and can directly use functions in library. The first part is to learn how to write C programs, and the second part is to learn how to use functions in C standard library.
- Object-oriented programming (C++ or Java) focuses on objects (nouns) rather than actions (verbs) (C or Pascal). Its basic entity is called **class**. The **reusability** has been greatly enhanced, which is beneficial for large software projects.
- C Programs typically go through six phases to be executed: **edit, preprocess, compile, link, load and execute**.
- Integrated Developing Environment (IDE) such as Microsoft Visual C++ 2005 Express Edition Software provides edit, preprocess, compile and link in an integrated environment.

35

Web Resources for Self-learning

- Google: http://www.google.com.hk/
- Wikipedia: http://en.wikipedia.org/wiki/Main_Page
- C programming: http://www.cprogramming.com/

36

The End

- Thank you very much!

37

Lecture 2

Introduction to C Programming

The C language facilitates a structured and disciplined approach to computer program design.

Acknowledgment

The notes are adapted from those provided by Deitel & Associates, Inc. and Pearson Education Inc.

38

OBJECTIVES

In this chapter you will learn:

- Print a line
- Escape sequence
- Variables and Data types
- Memory concepts
- Arithmetic operators
- If-then statements
- Review

39

A Simple C Program: Printing a Line of Text

```
1  /* Fig. 2.1: fig02_01.c
2     A first program in C */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main( void )
7  {
8     printf( "Welcome to C!\n" );
9
10    return 0; /* indicate that program ended successfully */
11
12 } /* end function main */
Welcome to C!
```

- `/*` and `*/` indicate comments – ignored by compiler
- `#include` directive tells C to load a particular file
- Left brace declares beginning of `main` function
- Statement tells C to perform an action
- `return` statement ends the function
- Right brace declares end of `main` function

fig02_01.c

- Comments
  - Text surrounded by /* and */ is ignored by computer
  - Used to describe program
- #include <stdio.h>
  - A directive to the C preprocessor
    - Tells computer to load contents of a certain file
  - `<stdio.h>` allows standard input/output operations

40

A Simple C Program: Printing a Line of Text

- `int main()`
  - C++ programs contain one or more functions, exactly one of which must be `main`
  - Parenthesis used to indicate a function
  - `int` means that `main` "returns" an integer value
  - Braces (`{` and `}`) indicate a block
    - The bodies of all functions must be contained in braces
- `printf( "Welcome to C!\n" );`
  - Instructs computer to perform an action
    - Specifically, prints the string of characters within quotes (" ")
  - Entire line called a statement
    - All statements must end with a semicolon (`;`)
  - Escape character (\)
    - Indicates that printf should do something out of the ordinary
    - `\n` is the newline character

41

A Simple C Program: Printing a Line of Text

- `return 0;`
  - A way to exit a function
  - `return 0`, in this case, means that the program terminated normally
- Right brace `}`
  - Indicates end of `main` has been reached
- Linker
  - When a function is called, linker locates it in the library
  - Inserts it into object program
  - If function name is misspelled, the linker will produce an error because it will not be able to find function in the library

42

OBJECTIVES

In this chapter you will learn:

- Print a line
- Escape sequence
- Variables and Data types
- Memory concepts
- Arithmetic operators
- If-then statements
- Review

43

Some common escape sequences

★

Escape sequence	Description
`\n`	Newline. Position the cursor at the beginning of the next line.
`\t`	Horizontal tab. Move the cursor to the next tab stop.
`\a`	Alert. Sound the system bell.
`\\`	Backslash. Insert a backslash character in a string.
`\"`	Double quote. Insert a double-quote character in a string.

44

The printf function can print Welcome to C! several different ways

fig02_03.c

```c
/* Fig. 2.3: fig02_03.c
   Printing on one line with two printf statements */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   printf( "Welcome " );
   printf( "to C!\n" );

   return 0; /* indicate that program ended successfully */

} /* end function main */
```

printf statement starts printing from where the last statement ended, so the text is printed on one line.

```
Welcome to C!
```

45

The printf function can print Welcome to C! several different ways

fig02_04.c

```c
/* Fig. 2.4: fig02_04.c
   Printing multiple lines with a single printf */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   printf( "Welcome\nto\nC!\n" );

   return 0; /* indicate that program ended successfully */

} /* end function main */
```

Newline characters move the cursor to the next line

```
Welcome
to
C!
```

46

OBJECTIVES

In this chapter you will learn:

- Print a line
- Escape sequence
- Variables and Data types
- Memory concepts
- Arithmetic operators
- If-then statements
- Review

47

Another Simple C Program: Adding Two Integers

- As before
  - Comments, `#include <stdio.h>` and `main`
- `int integer1, integer2, sum;`
  - Definition of variables
    - Variables: locations in memory where a value can be stored
  - `int` means the variables can hold integers (`-1`, `3`, `0`, `47`)
  - Variable names (identifiers)
    - `integer1`, `integer2`, `sum`
    - Identifiers: consist of letters, digits (cannot begin with a digit) and underscores(_)
      - Case sensitive
  - ★Definitions appear before executable statements
    - If an executable statement references and undeclared variable it will produce a syntax (compiler) error

48

Another Simple C Program: Adding Two Integers

- `scanf( "%d", &integer1 );`
  - Obtains a value from the user
    - `scanf` uses standard input (usually keyboard)
  - This `scanf` statement has two arguments
    - `%d` - indicates data should be a decimal integer
    - `&integer1` - location in memory to store variable
    - `&` is confusing in beginning – for now, just remember to include it with the variable name in `scanf` statements
  - When executing the program the user responds to the `scanf` statement by typing in a number, then pressing the *enter* (return) key

49

OBJECTIVES

In this chapter you will learn:

- Print a line
- Escape sequence
- Variables and Data types
- Memory concepts
- Arithmetic operators
- If-then statements
- Review

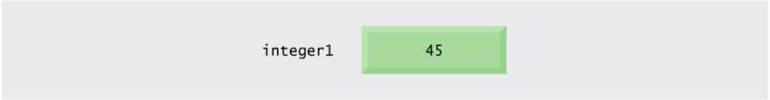
50

Memory Concepts

★ Variables

- Variable names correspond to locations in the computer's memory
- Every variable has a name, a type, a size and a value
- Whenever a new value is placed into a variable (through `scanf`, for example), it replaces (and destroys) the previous value
- Reading variables from memory does not change them

Fig. 2.6 | Memory location showing the name and value of a variable.

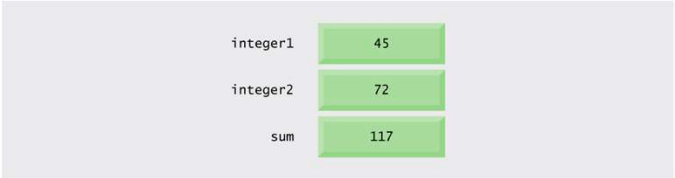


51

Memory Concepts

Fig. 2.7 | Memory locations after both variables are input.

Fig. 2.8 | Memory locations after a calculation.



52

OBJECTIVES

In this chapter you will learn:

- Print a line
- Escape sequence
- Variables and Data types
- Memory concepts
- Arithmetic operators
- If-then statements
- Review

53

Another Simple C Program: Adding Two Integers

- **=** (assignment operator)
  - Assigns a value to a variable
  - Is a binary operator (has two operands)
    `sum = variable1 + variable2;`
    `sum gets variable1 + variable2;`
  - Variable receiving value on left
- `printf( "Sum is %d\n", sum );`
  - Similar to `scanf`
    - `%d` means decimal integer will be printed
    - `sum` specifies what integer will be printed
  - Calculations can be performed inside `printf` statements
    `printf( "Sum is %d\n", integer1 + integer2 );`

54

Arithmetic

- Arithmetic calculations
  - Use * for multiplication and / for division
  - Integer division truncates remainder
    - 7 / 5 evaluates to 1
  - Modulus operator(%) returns the remainder
    - 7 % 5 evaluates to 2
- Operator precedence
  - Some arithmetic operators act before others (i.e., multiplication before addition)
    - Use parenthesis when needed
  - Example: Find the average of three variables a, b and c
    - Do not use: a + b + c / 3
    - Use: (a + b + c) / 3

55

Arithmetic ★

- Arithmetic operators:

C operation	Arithmetic operator	Algebraic expression	C expression
Addition	+	$f + 7$	f + 7
Subtraction	-	$p - c$	p - c
Multiplication	*	bm	b * m
Division	/	x / y	x / y
Modulus	%	$r \bmod s$	r % s

- Rules of operator precedence:

Operator(s)	Operation(s)	Order of evaluation (precedence)
O	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
*, /, or %	Multiplication,Division, Modulus	Evaluated second. If there are several, they are evaluated left to right.
+ or -	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

56

Decision Making: Equality and Relational Operators

Step 1.	`y = 2 * 5 * 5 + 3 * 5 + 7;` `2 * 5 is 10`	*(Leftmost multiplication)*
Step 2.	`y = 10 * 5 + 3 * 5 + 7;` `10 * 5 is 50`	*(Leftmost multiplication)*
Step 3.	`y = 50 + 3 * 5 + 7;` `3 * 5 is 15`	*(Multiplication before addition)*
Step 4.	`y = 50 + 15 + 7;` `50 + 15 is 65`	*(Leftmost addition)*
Step 5.	`y = 65 + 7;` `65 + 7 is 72`	*(Last addition)*
Step 6.	`y = 72`	*(Last operation—place 72 in y)*

57

```c
/* Fig. 2.5: fig02_05.c
   Addition program */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int integer1; /* first number to be input by user  */
   int integer2; /* second number to be input by user */
   int sum;      /* variable in which sum will be stored */

   printf( "Enter first integer\n" ); /* prompt */
   scanf( "%d", &integer1 );          /* read an integer */

   printf( "Enter second integer\n" ); /* prompt */
   scanf( "%d", &integer2 );           /* read an integer */

   sum = integer1 + integer2; /* assign total to sum */

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

   return 0;  /* indicate that program ended successfully */

} /* end function main */
```

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

fig02_05.c

Definitions of variables

scanf obtains a value from the user and assigns it to **integer1**

scanf obtains a value from the user and assigns it to **integer2**

Assigns a value to **sum**

Another Simple C Program: Adding Two Integers

58

OBJECTIVES

In this chapter you will learn:

- Print a line
- Escape sequence
- Variables and Data types
- Memory concepts
- Arithmetic operators
- If-then statements
- Review

59

Decision Making: Equality and Relational Operators

- Executable statements
  - Perform actions (calculations, input/output of data)
  - Perform decisions
    - May want to print `"pass"` or `"fail"` given the value of a test grade
- `if` control statement
  - If a condition is `true`, then the body of the `if` statement executed
    - `0` is `false`, non-zero is `true`
  - Control always resumes after the `if` structure
- Keywords
  - Special words reserved for C
  - Cannot be used as identifiers or variable names

60

Decision Making: Equality and Relational Operators

Standard algebraic equality operator or relational operator	C equality or relational operator	Example of C condition	Meaning of C condition
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

61

An Example to Use if statements, relational operators, and equality operators

```c
/* Fig. 2.13: fig02_13.c
   Using if statements, relational
   operators, and equality operators */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int num1; /* first number to be read from user  */
   int num2; /* second number to be read from user */

   printf( "Enter two integers, and I will tell you\n" );
   printf( "the relationships they satisfy: " );

   scanf( "%d%d", &num1, &num2 ); /* read two integers */

   if ( num1 == num2 ) {
      printf( "%d is equal to %d\n", num1, num2 );
   } /* end if */

   if ( num1 != num2 ) {
      printf( "%d is not equal to %d\n", num1, num2 );
   } /* end if */

   if ( num1 < num2 ) {
      printf( "%d is less than %d\n", num1, num2 );
   } /* end if */

```

Checks if **num1** is equal to **num2**

Checks if **num1** is not equal to **num2**

Checks if **num1** is less than **num2**

fig02_13.c
(1 of 3)

62

```
   if ( num1 > num2 ) {
      printf( "%d is greater than %d\n", num1, num2 );
   } /* end if */

   if ( num1 <= num2 ) {
      printf( "%d is less than or equal to %d\n", num1, num2 );
   } /* end if */

   if ( num1 >= num2 ) {
      printf( "%d is greater than or equal to %d\n", num1, num2 );
   } /* end if */

   return 0;   /* indicate that program ended successfully */

} /* end function main */
} /* end function main */
```

Checks if **num1** is greater than **num2**

Checks if **num1** is less than or equal to **num2**

Checks if **num1** is greater than equal to **num2**

```
Enter two integers, and I will tell you
the relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7
```

(continued on next slide…)

fig02_13.c (2 of 3)

63

(continued from previous slide…)

```
Enter two integers, and I will tell you
the relationships they satisfy:
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12
```

```
Enter two integers, and I will tell you
the relationships they satisfy:
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

fig02_13.c (3 of 3)

64

Fig. 2.14 | Precedence and associativity of the operators discussed so far.

Operators	Associativity
`( )`	left to right
`*  /  %`	left to right
`+  -`	left to right
`<  <=  >  >=`	left to right
`==  !=`	left to right
`=`	right to left

Fig. 2.15 | C's keywords.

Keywords			
`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

65

OBJECTIVES

In this chapter you will learn:

- Print a line
- Escape sequence
- Variables and Data types
- Memory concepts
- Arithmetic operators
- If-then statements
- Review

66

Review

- ★ Comments begin with /* and end with */. Comments document programs and improve readability.
- ★ The #include directive tells the preprocessor to include the content of another file (typically a header file such as <stdio.h>).
  - The <stdio.h> header contains information used by the compiler when compiling calls to standard input/output library functions such as printf.
- ★ Every program in C begins executing at the function main.
  - Functions can return information.
  - Functions can receive information.
  - A function starts by a left brace { and ends by a right brace }.
  - Every statement must end by a semicolon.
- ★ Escape sequences, for example \n newline.

67

Review

- A variable is a location in memory where a value can be stored for use by a program.
- ★ All variables must be defined with a name and a data type before they can be used in a program.
  - The %d indicates that the data should be an decimal integer.
  - The & indicates the memory address of an variable.
  - Function printf can also use a format control string.
- ★ Every variable has a name, a type and a value. Variable names correspond to locations in the computer's memory.
- ★ Arithmetic operators and precedence.

68

Review

- ★ If statement allows a program to make a decision based on the truth or falsity of a statement of fact called a condition.
- Conditions in if statement are formed by suing the equality operators and relational operators.
- ★ The precedence of relational operators
- = should be read “gets” and == should be read “double equals”.
- Key words.

69

The End

- Thank you very much!

70

Lecture 3

Structured Program Development in C

Before writing a program to solve a particular problem, it is essential to have a thorough understanding of the problem and a carefully planned approach to solving the problem.

Acknowledgment

The notes are adapted from those provided by Deitel & Associates, Inc. and Pearson Education Inc.

71

OBJECTIVES

- In this lecture, you will learn:
  - Develop algorithms
    - Psudocode and flowchart
  - Control structures
    - Sequence structures
    - Selection structures (`if` selection statement and `if…else` selection statement)
    - Repetition structures (`while` repetition statement to execute statements in a program repeatedly)
  - Structured programming with Top-Down refinement
  - The assignment, increment and decrement operators
  - Review

72

Algorithms

- Computing problems --- Target of Algorithms
  - All can be solved by executing a series of actions in a specific order
- Algorithm: procedure in terms of
  - Actions to be executed
  - The order in which these actions are to be executed
- Program control
  - Specify order in which statements are to be executed
- Two representations of an algorithm
  - Pseudocode --- style like everyday's English
  - Flowchart --- a graphical representation

73

Pseudocode

★ 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
  - Consist only of executable statements, i.e. without comments and variable definition.

74

Flowchart

- 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

75

OBJECTIVES

- In this lecture, you will learn:
  - Develop algorithms
    - Psudocode and flowchart
  - Control structures
    - Sequence structures
    - Selection structures (`if` selection statement and `if`…`else` selection statement)
    - Repetition structures (`while` repetition statement to execute statements in a program repeatedly)
  - Structured programming with Top-Down refinement
  - The assignment, increment and decrement operators
  - Review

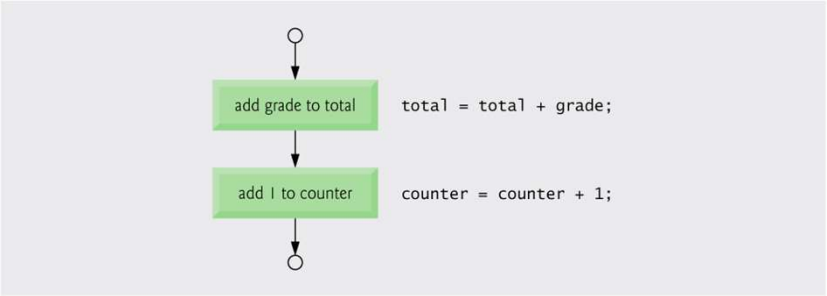
76

Control Structures

- Sequential execution (by default)
  - Statements executed one after the other in the order written
- Transfer of control
  - When the next statement executed is not the next one in sequence
  - Overuse of `goto` statements led to many problems
- All programs written in terms of three control structures
  - Sequence structures: Built into C. Programs executed sequentially by default
  - Selection structures: C has three types: `if`, `if…else`, and `switch`
  - Repetition structures: C has three types: `while`, `do…while` and `for`

77

Flowcharting C's sequence structure



78

The `if` Selection Statement

- Selection structure:
  - Used to choose among alternative courses of action
  - Pseudocode:

    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

79

The `if` Selection Statement

- Pseudocode statement in C:

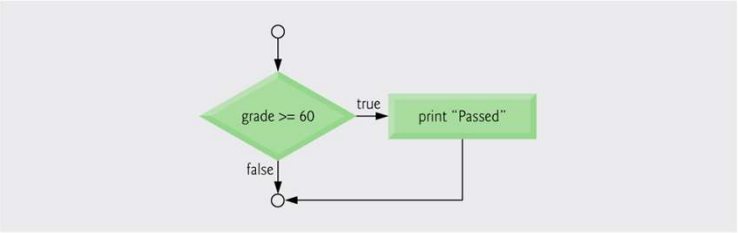
```
if ( grade >= 60 )
   printf( "Passed\n" );
```

  - 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

80

The `if` Selection Statement

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

81

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:

  If student's grade is greater than or equal to 60
  Print "Passed"
  else
  Print "Failed"

  - Note spacing/indentation conventions

82

The `if…else` Selection Statement

- C code:

```
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:

```
printf( "%s\n", grade >= 60 ? "Passed" :
   "Failed" );
```

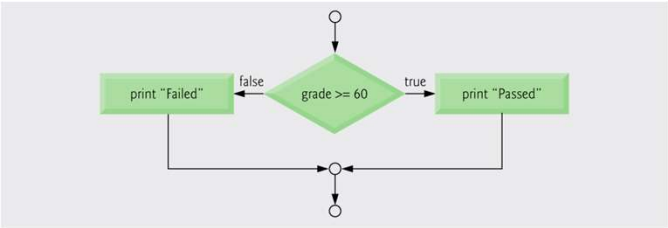
  - Or it could have been written:

```
grade >= 60 ? printf( “Passed\n” ) : printf(
   “Failed\n” );
```

83

The `if…else` Selection Statement

- Flow chart of the `if…else` selection statement



- 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

84

The `if…else` Selection Statement

- Pseudocode for a nested `if…else` statement

```
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"
```

85

The `if…else` Selection Statement

- Compound statement:
  - Set of statements within a pair of braces
  - Example:

```
if ( grade >= 60 )
   printf( "Passed.\n" );
else {
   printf( "Failed.\n" );
   printf( "You must take this course
      again.\n" );
  }
```

  - Without the braces, the statement

```
printf( "You must take this course
   again.\n" );
```

  would be executed automatically

86

Programming Errors in `if…else` Selection Statement

- Syntax errors
  - E.g. Forget one or both of the braces that delimit a compound statement
  - Caught by compiler
- Logic errors:
  - Have their effect at execution time
  - Non-fatal: program runs, but has incorrect output
  - Fatal: program exits prematurely

87

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`

88

The `while` Repetition Statement

- Example:

```
int product = 2;
while ( product <= 1000 )
       product = 2 * product;
```

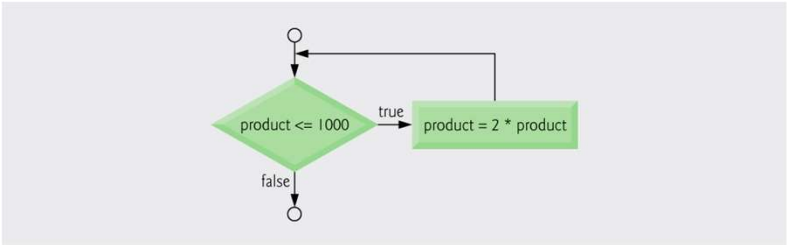


Fig. 3.4 | Flowcharting the **`while`** repetition statement.

89

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

90

```c
/* Fig. 3.6: fig03_06.c
   Class average program with counter-controlled repetition */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   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 */
```

- fig03_06.c (1 of 2)

Counter to control **while** loop

Initialize **counter** to 1

while loop iterates as long as **counter <= 10**

Increment the counter

91

```c

   /* termination phase */
   average = total / 10; /* integer division */

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

   return 0; /* indicate program ended successfully */

} /* end function main */
```

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

Calculate the average

- fig03_06.c (2 of 2)

92

OBJECTIVES

- In this lecture, you will learn:
  - Develop algorithms
    - Psudocode and flowchart
  - Control structures
    - Sequence structures
    - Selection structures (`if` selection statement and `if…else` selection statement)
    - Repetition structures (`while` repetition statement to execute statements in a program repeatedly)
  - Structured programming with Top-Down refinement
  - The assignment, increment and decrement operators
  - Review

93

Top-Down, Stepwise Refinement

- Problem becomes:

  Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.
  - Unknown number of students
  - How will the program know to end?
- Use sentinel value
  - Also called signal value, dummy value, or flag value
  - Indicates "end of data entry."
  - Loop ends when user inputs the sentinel value
  - Sentinel value chosen so it cannot be confused with a regular input (such as `-1` in this case)

94

Top-Down, Stepwise Refinement

- Top-down, stepwise refinement
  - Begin with a pseudocode representation of the *top*:

    Determine the class average for the quiz
  - Divide *top* into smaller tasks and list them in order:

    Initialize variables
    Input, sum and count the quiz grades
    Calculate and print the class average
- ★ Many programs have three phases:
  - Initialization: initializes the program variables
  - Processing: inputs data values and adjusts program variables accordingly
  - Termination: calculates and prints the final results

95

Notice!!

- Each refinement, as well as the top itself, is a complete specification of the algorithm; only the level of detail varies.

96

Top-Down, Stepwise Refinement

- Refine the initialization phase from *Initialize variables* to:

  Initialize total to zero
  Initialize counter to zero

- Refine *Input, sum and count the quiz grades* to

  Input the first grade (possibly the sentinel)
  While the user has not as yet entered the sentinel
  Add this grade into the running total
  Add one to the grade counter
  Input the next grade (possibly the sentinel)

97

Top-Down, Stepwise Refinement

- Refine *Calculate and print the class average* to

  If the counter is not equal to zero
  Set the average to the total divided by the counter
  Print the average
  else
  Print "No grades were entered"

98

Pseudocode algorithm that uses sentinel-controlled repetition to solve the class average problem

```
Initialize total to zero
Initialize counter to zero

Input the first grade
While the user has not as yet entered the sentinel
  Add this grade into the running total
  Add one to the grade counter
  Input the next grade (possibly the sentinel)

If the counter is not equal to zero
  Set the average to the total divided by the counter
  Print the average
else
  Print "No grades were entered"
```

99

When to stop the top-down, stepwise refinement process?

- Stop the top-down, stepwise refinement process when the pseudocode algorithm is specified in sufficient detail for you to be able to convert the pseudocode to C.
- Implementing the C program is then normally straightforward.

100

```c
/* Fig. 3.8: fig03_08.c
   Class average program with sentinel-controlled repetition */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int counter; /* number of grades entered */
   int grade;   /* grade value */
   int total;   /* sum of grades */

   float average; /* number with decimal point for average */

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

   /* processing phase */
   /* get first grade from user */
   printf( "Enter grade, -1 to end: " ); /* prompt for input */
   scanf( "%d", &grade );                /* read grade from user */

```

- fig03_08.c (1 of 3)

float type indicates variable can be a non-integer

101

```c
   /* loop while sentinel value not yet read from user */
   while ( grade != -1 ) {
      total = total + grade; /* add grade to total */
      counter = counter + 1; /* increment counter */

      /* get next grade from user */
      printf( "Enter grade, -1 to end: " ); /* prompt for input */
      scanf("%d", &grade);                  /* read next grade */
   } /* end while */

   /* termination phase */
   /* if user entered at least one grade */
   if ( counter != 0 ) {

      /* calculate average of all grades entered */
      average = ( float ) total / counter; /* avoid truncation */

      /* display average with two digits of precision */
      printf( "Class average is %.2f\n", average );
   } /* end if */
   else { /* if no grades were entered, output message */
      printf( "No grades were entered\n" );
   } /* end else */

   return 0; /* indicate program ended successfully */

} /* end function main */
```

- fig03_08.c (2 of 3)

while loop repeats until user enters a value of -1

Ensures the user entered at least one grade

Converts **total** to **float** type

Prints result with 2 digits after decimal point

(float) is the cast operator that changes the data type of a variable.

102

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

```
Enter grade, -1 to end: -1
No grades were entered
```

- fig03_08.c (3 of 3)

103

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

104

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

105

Nested control structures

- Refine *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 *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"

106

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. 3.9 | Pseudocode for examination results problem.

107

```c
/* Fig. 3.10: fig03_10.c
   Analysis of examination results */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   /* 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 */
      else { /* otherwise, increment failures */
         failures = failures + 1;
      } /* end else */

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

- fig03_10.c (1 of 3)

while loop continues until 10 students have been processed

if and **else** statements are nested inside **while** loop

108

```c

   /* 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): 2
Enter Result (1=pass,2=fail): 2
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): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Passed 6
Failed 4
```

(continued on next slide…)

- fig03_10.c (2 of 3)

109

(continued from previous slide…)

```
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): 2
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
```

- fig03_10.c (3 of 3)

110

OBJECTIVES

- In this lecture, you will learn:
  - Develop algorithms
    - Psudocode and flowchart
  - Control structures
    - Sequence structures
    - Selection structures (`if` selection statement and `if…else` selection statement)
    - Repetition structures (`while` repetition statement to execute statements in a program repeatedly)
  - Structured programming with Top-Down refinement
  - The assignment, increment and decrement operators
  - Review

111

Assignment Operators

- Assignment operators abbreviate assignment expressions

  `c = c + 3;`

  can be abbreviated as `c += 3;` using the addition assignment operator
- Statements of the form

  variable = *variable operator expression*;

  can be rewritten as

  variable `operator` = *expression*;
- Examples of other assignment operators:

```
d -= 4      (d = d - 4)
e *= 5      (e = e * 5)
f /= 3      (f = f / 3)
g %= 9      (g = g % 9)
```

112

Assignment Operators

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`C = c + 7`	10 to c
`-=`	`d -= 4`	`D = d - 4`	1 to d
`*=`	`e *= 5`	`E = e * 5`	20 to e
`/=`	`f /= 3`	`F = f / 3`	2 to f
`%=`	`g %= 9`	`G = g % 9`	3 to g

113

Increment and Decrement Operators

- Increment operator (++)
  - Can be used instead of `c+=1`
- Decrement operator (--)
  - Can be used instead of `c-=1`
- Preincrement
  - Operator is used before the variable (`++c` or `--c`)
  - Variable is changed before the expression it is in is evaluated
- Postincrement
  - Operator is used after the variable (`c++` or `c--`)
  - Expression executes before the variable is changed

114

Increment and Decrement Operators

- If `c` equals 5, then
  `printf( "%d", ++c );`
  - Prints 6
  `printf( "%d", c++ );`
  - Prints 5
  - In either case, `c` now has the value of 6
- When variable not in an expression
  - Preincrementing and postincrementing have the same effect
  `++c;`
  `printf( “%d”, c );`
  - Has the same effect as
  `c++;`
  `printf( “%d”, c );`

115

Increment and Decrement Operators

Operator	Sample expression	Explanation
++	++a	Increment **a** by 1, then use the new value of **a** in the expression in which **a** resides.
++	a++	Use the current value of **a** in the expression in which **a** resides, then increment **a** by 1.
--	--b	Decrement **b** by 1, then use the new value of **b** in the expression in which **b** resides.
--	b--	Use the current value of **b** in the expression in which **b** resides, then decrement **b** by 1.

++a; → a = a + 1; --b; → b = b – 1;

Unary operators should be placed directly next to their operands with no intervening spaces.

116

```c
/* Fig. 3.13: fig03_13.c
   Preincrementing and postincrementing */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int c;                  /* define variable */

   /* demonstrate postincrement */
   c = 5;                  /* assign 5 to c */
   printf( "%d\n", c );    /* print 5 */
   printf( "%d\n", c++ ); /* print 5 then postincrement */
   printf( "%d\n\n", c ); /* print 6 */

   /* demonstrate preincrement */
   c = 5;                  /* assign 5 to c */
   printf( "%d\n", c );    /* print 5 */
   printf( "%d\n", ++c ); /* preincrement then print 6 */
   printf( "%d\n", c );    /* print 6 */

   return 0; /* indicate program ended successfully */

} /* end function main */
```

```
5
5
6

5
6
6
```

- fig03_13.c

c is printed, then incremented

c is incremented, then printed

117

Precedence of the operators encountered so far in the text

Operators	Associativity	Type
`++` *(postfix)* `--` *(postfix)*	right to left	postfix
`+` `-` `(`*type*`)` `++` *(prefix)* `--` *(prefix)*	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment

118

OBJECTIVES

- In this lecture, you will learn:
  - Develop algorithms
    - Psudocode and flowchart
  - Control structures
    - Sequence structures
    - Selection structures (`if` selection statement and `if…else` selection statement)
    - Repetition structures (`while` repetition statement to execute statements in a program repeatedly)
  - Structured programming with Top-Down refinement
  - The assignment, increment and decrement operators
  - Review

119

Review

★ Psedocodes consists only of action statements.

- It is better to list each variable and briefly mention the purpose of each at the beginning of a pseudocode program.

★ Three types of control structures are sequence structures, selection structures and repetition structures.

★ A flowchart is a graphical representation of an algorithm. Flowcharts are drawn using certain special-purpose symbols such as rectangles (action symbol), diamonds (decision symbol), ovals, and small circles are connected by arrows called flowlines (the order of the actions).

- If selection, if else selection and switch selection statements.
- While, do… while and for statement in repetition structures.

120

Review

- To include several statements in the body of an IF, enclose the set of statements in {}.
- Counter-controlled repetition and Sentinel-controlled repetition.
- ★ Top-down, stepwise refinement is a technique that is essential to the development of well-structured program.
- The type float represents floating-point numbers.
- To produce a floating-point calculation with integer values, you must cast the integers to floating-point numbers.
- ★ Assignment operators.
  - a = a + b; → a += b;
- ★ Unary increment ++ and decrement operator --.

121

The End

- Thank you very much!

122

Lecture 4

C Program Control

Acknowledgment

The notes are adapted from those provided by Deitel & Associates, Inc. and Pearson Education Inc.

123

OBJECTIVES

In this lecture, you will learn:

- The `while`, `for` and `do...while` repetition statements to execute statements in a program repeatedly;
- The `break` and `continue` program control statements to alter the flow of program control;
- The `switch` selection statement;
- The logical operators to form complex conditional expressions in control statements;
- To avoid confusing the equality and assignment operators
- Structured-Programming Summary
- Review

124

Repetition Essentials

- 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"

125

Counter-Controlled Repetition

- 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)

126

Counter-Controlled Repetition

- Example:

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

  - The statement

```
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

127

```
1  /* Fig. 4.1: fig04_01.c
2     Counter-controlled repetition */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main( void )
7  {
8     int counter = 1; /* initialization */
9
10    while ( counter <= 10 ) { /* repetition condition */
11       printf ( "%d\n", counter ); /* display counter */
12       ++counter; /* increment */
13    } /* end while */
14
15    return 0; /* indicate program ended successfully */
16
17 } /* end function main */
```

Definition and assignment are performed simultaneously

```
1
2
3
4
5
6
7
8
9
10
```

- fig04_01.c

128

Counter-Controlled Repetition

- Condensed code
  - C Programmers would make the program more concise
  - Initialize `counter` to 0
    -
```
while ( ++counter <= 10 )
        printf( "%d\n, counter );
```

129

Important Notice!

★ Controlling counting loops with floating-point variables may result in imprecise counter values and inaccurate tests for termination

- Reason: floating-point values may be approximate.

130

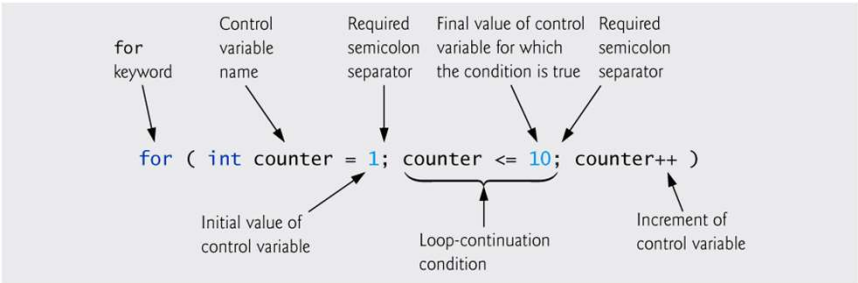
```c
/* Fig. 4.2: fig04_02.c
   Counter-controlled repetition with the for statement */
#include <stdio.h>

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

   /* initialization, repetition condition, and increment
      are all included in the for statement header. */
   for ( counter = 1; counter <= 10; counter++ ) {
      printf( "%d\n", counter );
   } /* end for */

   return 0; /* indicate program ended successfully */

} /* end function main */
```

- fig04_02.c

for loop begins by setting **counter** to 1 and repeats while **counter <= 10**. Each time the end of the loop is reached, **counter** is incremented by 1.

131

The for Repetition Statement



132

The `for` Repetition Statement

- Format when using `for` loops

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

- Example:

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

  - Prints the integers from one to ten

No semicolon (`;`) after last expression

133

The `for` Statement

- 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 );
```

134

The `for` Statement : Notes and Observations

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

    ```
    for ( j = x; j <= 4 * x * y; j += y / x )
    ```

  is equivalent to

    ```
    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

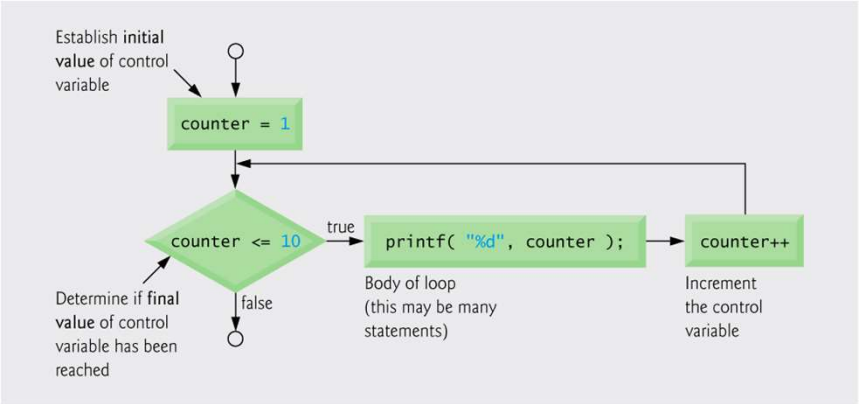
135

Notice!

★ Although the value of the control variable can be changed in the body of a `for` loop, this can lead to subtle errors. It is best not to change it.

136

Flowcharting a typical `for` repetition statement



137

```
1  /* Fig. 4.6: fig04_06.c
2     Calculating compound interest */
3  #include <stdio.h>
4  #include <math.h>
5
6  /* function main begins program execution */
7  int main( void )
8  {
9     double amount;             /* amount on deposit */
10    double principal = 1000.0; /* starting principal */
11    double rate = .05;         /* annual interest rate */
12    int year;                  /* year counter */
13
14    /* output table column head */
15    printf( "%4s%21s\n", "Year", "Amount on deposit" );
16
17    /* calculate amount on deposit for each of ten years */
18    for ( year = 1; year <= 10; year++ ) {
19
20       /* calculate new amount for specified year */
21       amount = principal * pow( 1.0 + rate, year );
22
23       /* output one table row */
24       printf( "%4d%21.2f\n", year, amount );
25    } /* end for */
26
27    return 0; /* indicate program ended successfully */
28
29 } /* end function main */
```

additional header

- fig04_06.c (1 of 2)

pow function calculates the value of the first argument raised to the power of the second argument; It takes two arguments of type double and returns a double value.

138

```
Year    Amount on deposit
   1             1050.00
   2             1102.50
   3             1157.63
   4             1215.51
   5             1276.28
   6             1340.10
   7             1407.10
   8             1477.46
   9             1551.33
  10             1628.89
```

- fig04_06.c (2 of 2)

139

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 );
```

140

The do…while Repetition Statement

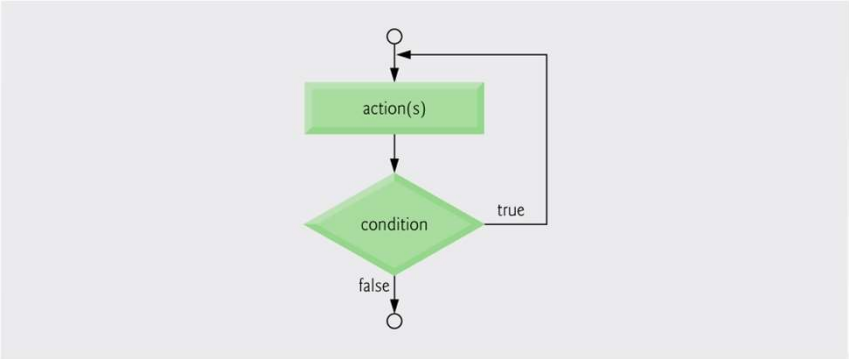
- Example (letting counter = 1):

```
do {
   printf( "%d  ", counter );
} while (++counter <= 10);
```

  - Prints the integers from 1 to 10

141

Flowchart of the do…while repetition statement



142

```c
/* Fig. 4.9: fig04_09.c
   Using the do/while repetition statement */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   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 */
```

```
1  2  3  4  5  6  7  8  9  10
```

- fig04_09.c

increments **counter** then checks if it is less than or equal to 10

143

OBJECTIVES

In this lecture, you will learn:

- The `while`, `for` and `do...while` repetition statements to execute statements in a program repeatedly;
- The `break` and `continue` program control statements to alter the flow of program control;
- The `switch` selection statement;
- The logical operators to form complex conditional expressions in control statements;
- To avoid confusing the equality and assignment operators
- Structured-Programming Summary
- Review

144

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

145

```c
/* Fig. 4.11: fig04_11.c
   Using the break statement in a for statement */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   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\n", x );

   return 0; /* indicate program ended successfully */

} /* end function main */
```

```
1 2 3 4
Broke out of loop at x == 5
```

break immediately ends **for** loop

- fig04_11.c

146

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

147

```c
/* Fig. 4.12: fig04_12.c
   Using the continue statement in a for statement */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   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 */
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

- fig04_12.c

`continue` skips to end of **`for`** loop and performs next iteration

148

OBJECTIVES

In this lecture, you will learn:

- The `while`, `for` and `do...while` repetition statements to execute statements in a program repeatedly;
- The `break` and `continue` program control statements to alter the flow of program control;
- The `switch` selection statement;
- The logical operators to form complex conditional expressions in control statements;
- To avoid confusing the equality and assignment operators
- Structured-Programming Summary
- Review

149

The `switch` Multiple-Selection Statement

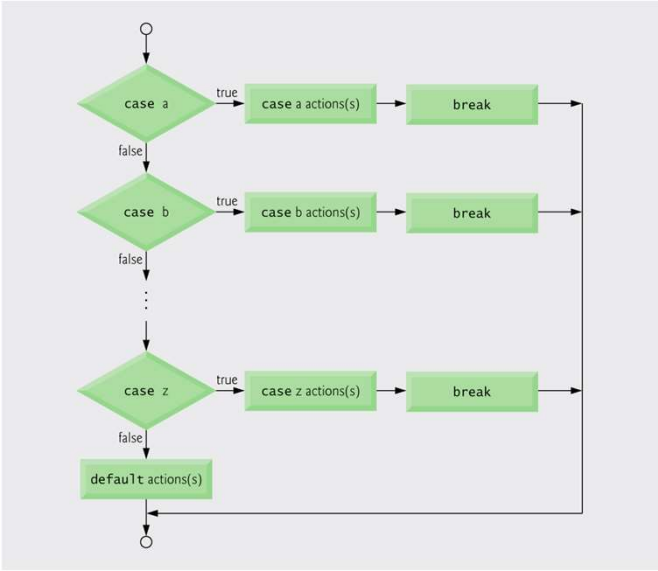
- `switch`
  - Useful when a variable or expression is tested for all the values it can assume and different actions are taken
- Format
  - Series of `case` labels and an optional `default` case

```
switch ( value ){
   case '1':
      actions
   case '2':
      actions
   default:
      actions
   }
```

★ `break;` exits from statement

150

Flowchart of the `switch` statement



151

```
1  /* Fig. 4.7: fig04_07.c
2     Counting letter grades */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main( void )
7  {
8     int grade;      /* one grade */
9     int aCount = 0; /* number of As */
10    int bCount = 0; /* number of Bs */
11    int cCount = 0; /* number of Cs */
12    int dCount = 0; /* number of Ds */
13    int fCount = 0; /* number of Fs */
14
15    printf(  "Enter the letter grades.\n"  );
16    printf(  "Enter the EOF character to end input.\n"  );
17
18    /* loop until user types end-of-file key sequence */
19    while ( ( grade = getchar() ) != EOF ) {
20
21       /* determine which grade was input */
22       switch ( grade ) { /* switch nested in while */
23
24          case 'A': /* grade was uppercase A */
25          case 'a': /* or lowercase a */
26             ++aCount; /* increment aCount */
27             break; /* necessary to exit switch */
28
```

- fig04_07.c (1 of 4)

EOF stands for "end of file;" this character varies from system to system

switch statement checks each of its nested **case**s for a match

break statement makes program skip to end of **switch**

152

```c
         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 */

```

- fig04_07.c (2 of 4)

153

```c
         default: /* catch all other characters */
            printf( "Incorrect letter grade entered." );
            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 */
```

default case occurs if none of the **case**s are matched

- fig04_07.c (3 of 4)

154

```
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
```

- fig04_07.c (4 of 4)

155

OBJECTIVES

In this lecture, you will learn:

- The `while`, `for` and `do...while` repetition statements to execute statements in a program repeatedly;
- The `break` and `continue` program control statements to alter the flow of program control;
- The `switch` selection statement;
- The logical operators to form complex conditional expressions in control statements;
- To avoid confusing the equality and assignment operators
- Structured-Programming Summary
- Review

156

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

Expression	Result
`true && false`	`false`
`true || false`	`true`
`!false`	`true`

157

Logical Operators

expression1	expression2	expression1 && expression2
0	0	0
0	nonzero	0
nonzero	0	0
nonzero	nonzero	1

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

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.

expression	!expression
0	1
nonzero	0

Fig. 4.15 | Truth table for operator ! (logical negation).

158

Operator Precedence and Associativity

Operators	Associativity	Type
`++` *(postfix)* `--` *(postfix)*	right to left	postfix
`+` `-` `!` `++` *(prefix)* `--` *(prefix)* (*type*)	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

159

OBJECTIVES

In this lecture, you will learn:

- The `while`, `for` and `do...while` repetition statements to execute statements in a program repeatedly;
- The `break` and `continue` program control statements to alter the flow of program control;
- The `switch` selection statement;
- The logical operators to form complex conditional expressions in control statements;
- To avoid confusing the equality and assignment operators
- Structured-Programming Summary
- Review

160

Confusing Equality (==) and Assignment (=) Operators

- 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 ==:

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

    - Checks `payCode`, if it is 4 then a bonus is awarded

161

Confusing Equality (==) and Assignment (=) Operators

- Example, replacing == with =:

```
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

162

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;`

163

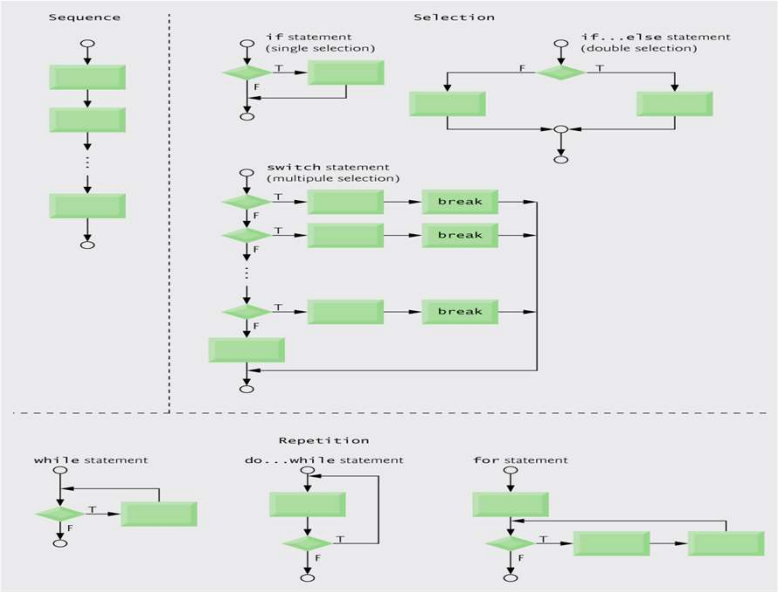
OBJECTIVES

In this lecture, you will learn:

- The `while`, `for` and `do...while` repetition statements to execute statements in a program repeatedly;
- The `break` and `continue` program control statements to alter the flow of program control;
- The `switch` selection statement;
- The logical operators to form complex conditional expressions in control statements;
- To avoid confusing the equality and assignment operators
- Structured-Programming Summary
- Review

164

Structured-Programming Summary



165

Structured-Programming Summary

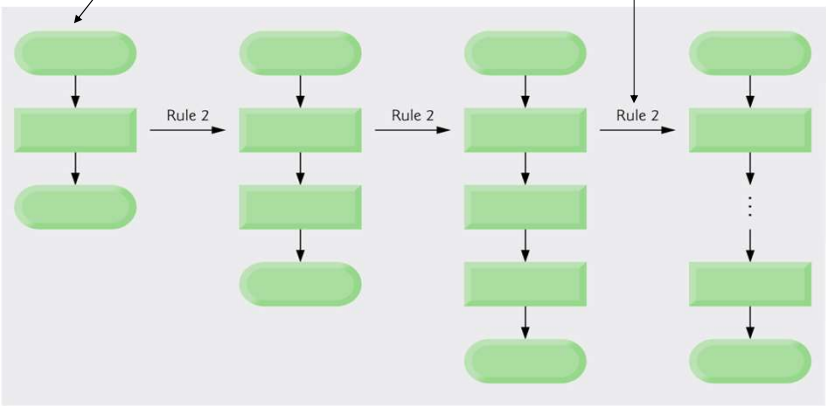
- Structured programming
  - Easier than unstructured programs to understand, test, debug and, modify programs
- Rules for structured programming
  - Rules developed by programming community
  - Only single-entry/single-exit control structures are used
  - Rules:
    1. Begin with the "simplest flowchart"
    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

166

Structured-Programming Summary

Rule 1 - Begin with the simplest flowchart

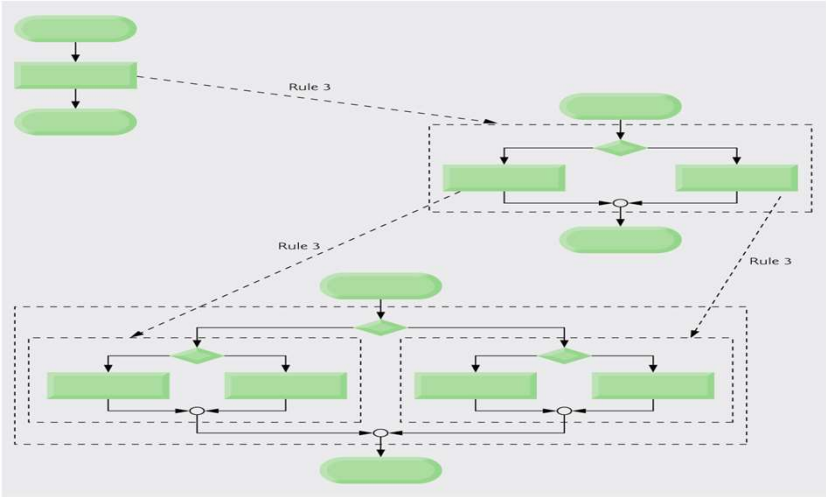
Rule 2 - Any rectangle can be replaced by two rectangles in sequence



167

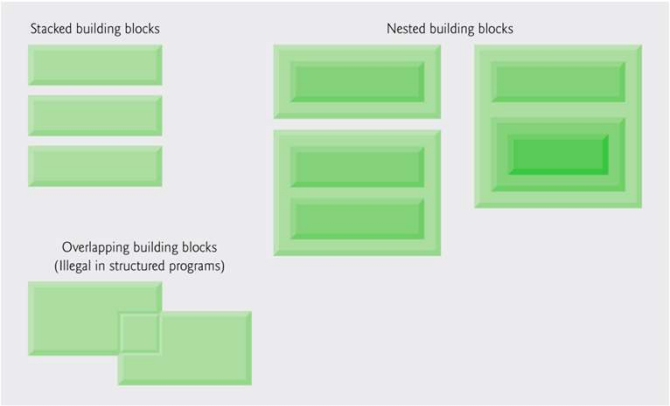
Structured-Programming Summary

Rule 3 - Replace any rectangle with a control structure



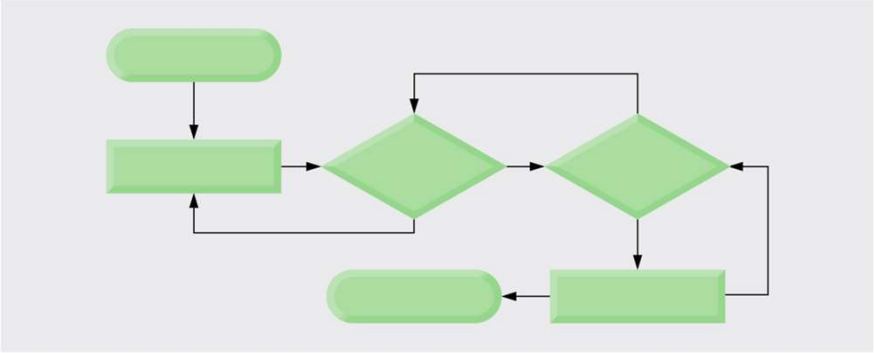
168

Stacked, nested and overlapped building blocks



169

An unstructured flowchart



170

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

171

OBJECTIVES

In this lecture, you will learn:

- The `while`, `for` and `do...while` repetition statements to execute statements in a program repeatedly;
- The `break` and `continue` program control statements to alter the flow of program control;
- The `switch` selection statement;
- The logical operators to form complex conditional expressions in control statements;
- To avoid confusing the equality and assignment operators
- Structured-Programming Summary
- Review

172

Review

- Counter-controlled and sentinel-controlled repetitions.
- ★ The `for` repetition statement handles all details of counter-controlled repetition.
- In most cases, `for` repetition statement can be represented with an equivalent `while` repetition statement.
- Initialization, loop-continuation condition and increment (or decrement).
- Data type `double` is a floating-point type much like `float`.
- The conversion specifier `%21.2f` denotes that a floating-point value will be displayed right 21 characters with two digits to the right of the decimal point.
- ★ `break` and `continue` statements.

173

Review

- ★ `switch` multiple selection statement.
- `do…while` repetition statement.
- ★ Logical operators `&&` `||` `!`
- Equality operator `==` and assignment operator `=`

174

The End

- Thank you very much!

175

Chapter 5

C Functions

The best way to develop and maintain a large program is to divide it into several smaller program modules, each of which is more manageable than the original program. Modules are written as functions in C.

Acknowledgment

The notes are adapted from those provided by Deitel & Associates, Inc. and Pearson Education Inc.

176

OBJECTIVES

- To construct programs modularly from small pieces called functions.
- The common math functions available in the C Standard Library.
- How to create new functions.
- The mechanisms used to pass information between functions.
- How to write and use recursive functions, i.e., functions that call themselves.

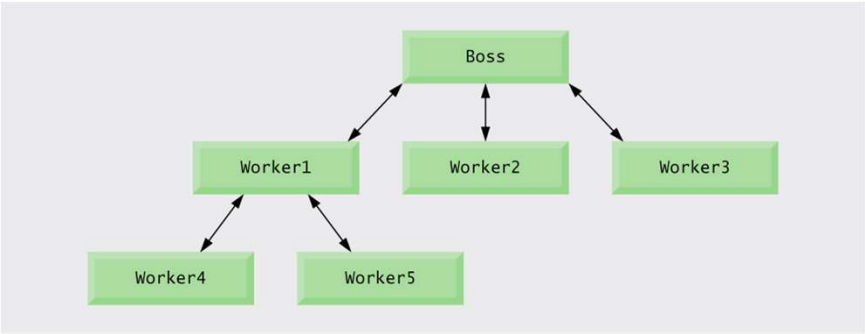
177

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

178

Hierarchical boss function/worker function relationship



179

OBJECTIVES

- To construct programs modularly from small pieces called functions.
- The common math functions available in the C Standard Library.
- How to create new functions.
- The mechanisms used to pass information between functions.
- How to write and use recursive functions, i.e., functions that call themselves.

180

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

181

Commonly used math library functions

Function	Description	Example
`sqrt( x )`	square root of x	`sqrt( 900.0 ) is 30.0` `sqrt( 9.0 ) is 3.0`
`exp( x )`	exponential function e^x	`exp( 1.0 ) is 2.718282` `exp( 2.0 ) is 7.389056`
`log( x )`	natural logarithm of x (base e)	`log( 2.718282 ) is 1.0` `log( 7.389056 ) is 2.0`

`log10( x )`	logarithm of x (base 10)	`log10( 1.0 ) is 0.0`
		`log10( 10.0 ) is 1.0`
		`log10( 100.0 ) is 2.0`

`fabs( x )`	absolute value of x	`fabs( 5.0 ) is 5.0`
		`fabs( 0.0 ) is 0.0`
		`fabs( -5.0 ) is 5.0`

`ceil( x )`	rounds x to the smallest integer not less than x	`ceil( 9.2 ) is 10.0` `ceil( -9.8 ) is -9.0`
`floor( x )`	rounds x to the largest integer not greater than x	`floor( 9.2 ) is 9.0` `floor( -9.8 ) is -10.0`
`pow( x, y )`	x raised to power y (x^y)	`pow( 2, 7 ) is 128.0` `pow( 9, .5 ) is 3.0`
`fmod( x, y )`	remainder of x/y as a floating point number	`fmod( 13.657, 2.333 ) is 1.992`
`sin( x )`	trigonometric sine of x (x in radians)	`sin( 0.0 ) is 0.0`
`cos( x )`	trigonometric cosine of x (x in radians)	`cos( 0.0 ) is 1.0`
`tan( x )`	trigonometric tangent of x (x in radians)	`tan( 0.0 ) is 0.0`

182

OBJECTIVES

- To construct programs modularly from small pieces called functions.
- The common math functions available in the C Standard Library.
- How to create new functions.
- The mechanisms used to pass information between functions.
- How to write and use recursive functions, i.e., functions that call themselves.

183

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

184

Tips

- In programs containing many functions, `main` is often implemented as a group of calls to functions that perform the bulk of the program's work.
- Each function should be limited to performing a single, well-defined task, and the function name should effectively express that task.

185

Function Definitions

- 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`
  - 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* `;`

186

- fig05_03.c

```c
/* 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( void )
{
   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 parameter */
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
```

Function prototype indicates function will be defined later in the program

Call to **square** function

Function definition

187

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`s
    - Returns an `int`
- Promotion rules and conversions
  - Converting to lower types can lead to errors

188

Promotion Hierarchy for Data Types

High

Low

Data type	printf conversion specification	scanf conversion specification
Long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
Unsigned long int	%lu	%lu
long int	%ld	%ld
unsigned int	%u	%u
int	%d	%d
unsigned short	%hu	%hu
short	%hd	%hd
char	%c	%c

189

```c
/* 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( void )
{
   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 prototype

Function call

- fig05_04.c (1 of 2)

190

```c
/* 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 */
```

Function definition

```
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
```

- fig05_04.c (2 of 2)

191

Notice!

- A function prototype tells the compiler:
  - the type of data return by the function
  - the number of parameters the function expects to receive
  - the types of the parameters, and
  - the order in which these parameters are expected
  - Prototype only needed if function definition comes after use in program
- By default, the compiler assumes the function returns an *int*, and nothing is assumed about the arguments.
- Omitting the return-value-type in a function definition is a syntax error if the function prototype specifies a return type other than *int*.
- Forgetting to return a value from a function that is supposed to return a value can lead to unexpected errors. The C standard states that the result of this omission is undefined.
- Returning a value from a function with a `void` return type is a syntax error.
- The function prototype, function header and function calls should all agree in the number, type, and order of arguments and parameters, and in the type of return value.

192

Header Files

- Header files
  - Contain function prototypes for library functions
  - `<stdlib.h>` , `<math.h>` , etc
  - Load with `#include <filename>`
    `#include <math.h>`
- Custom header files
  - Create file with functions
  - Save as `filename.h`
  - Load in other files with `#include "filename.h"`
  - Reuse functions

193

Some of the standard library header

Standard library header	Explanation
`<assert.h>`	Contains macros and information for adding diagnostics that aid program debugging.
`<ctype.h>`	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
`<errno.h>`	Defines macros that are useful for reporting error conditions.
`<float.h>`	Contains the floating point size limits of the system.
`<limits.h>`	Contains the integral size limits of the system.
`<locale.h>`	Contains function prototypes and other information that enables a program to be modified for the current locale on which it is running. The notion of locale enables the computer system to handle different conventions for expressing data like dates, times, dollar amounts and large numbers throughout the world.
`<math.h>`	Contains function prototypes for math library functions.
`<setjmp.h>`	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.
`<signal.h>`	Contains function prototypes and macros to handle various conditions that may arise during program execution.
`<stdarg.h>`	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
`<stddef.h>`	Contains common definitions of types used by C for performing certain calculations.
`<stdio.h>`	Contains function prototypes for the standard input/output library functions, and information used by them.
`<stdlib.h>`	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers, and other utility functions.
`<string.h>`	Contains function prototypes for string processing functions.
`<time.h>`	Contains function prototypes and types for manipulating the time and date.

194

OBJECTIVES

- To construct programs modularly from small pieces called functions.
- The common math functions available in the C Standard Library.
- How to create new functions.
- The mechanisms used to pass information between functions.
- How to write and use recursive functions, i.e., functions that call themselves.

195

Calling Functions: Call by Value and Call by Reference

- Call by value
  - Copy of argument passed to function
  - Changes in function do not effect original
  - Use when function does not need to modify argument
    - Avoids accidental changes
- Call by reference
  - Passes original argument
  - Changes in function effect original
  - Only used with trusted functions
- For now, we focus on call by value

196

Random Number Generation

- `rand` function
  - Load `<stdlib.h>`
  - Returns "random" number between `0` and `RAND_MAX` (at least `32767`)

    `i = rand();`
  - Pseudorandom
    - Preset sequence of "random" numbers
    - Same sequence for every function call
- Scaling
  - To get a random number between `1` and `n`

    `1 + ( rand() % n )`
    - `rand() % n` returns a number between `0` and `n - 1`
    - Add `1` to make random number between `1` and `n`

      `1 + ( rand() % 6)`
      - number between `1` and `6`

197

```c
/* Fig. 5.7: fig05_07.c
   Shifted, scaled integers produced by 1 + rand() % 6 */
#include <stdio.h>
#include <stdlib.h>

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

   /* loop 20 times */
   for ( i = 1; i <= 20; i++ ) {

      /* pick random number from 1 to 6 and output it */
      printf( "%10d", 1 + ( rand() % 6 ) );

      /* if counter is divisible by 5, begin new line of output */
      if ( i % 5 == 0 ) {
         printf( "\n" );
      } /* end if */

   } /* end for */

   return 0; /* indicates successful termination */

} /* end main */
```

Generates a random number between 1 and 6

- fig05_07.c

```
         6         6         5         5         6
         5         1         1         5         3
         6         6         2         4         2
         6         2         3         4         1
```

198

```c
/* Fig. 5.8: fig05_08.c
   Roll a six-sided die 6000 times */
#include <stdio.h>
#include <stdlib.h>

/* function main begins program execution */
int main( void )
{
   int frequency1 = 0; /* rolled 1 counter */
   int frequency2 = 0; /* rolled 2 counter */
   int frequency3 = 0; /* rolled 3 counter */
   int frequency4 = 0; /* rolled 4 counter */
   int frequency5 = 0; /* rolled 5 counter */
   int frequency6 = 0; /* rolled 6 counter */

   int roll; /* roll counter, value 1 to 6000 */
   int face; /* represents one roll of the die, value 1 to 6 */

   /* loop 6000 times and summarize results */
   for ( roll = 1; roll <= 6000; roll++ ) {
      face = 1 + rand() % 6; /* random number from 1 to 6 */

      /* determine face value and increment appropriate counter */
      switch ( face ) {

         case 1: /* rolled 1 */
            ++frequency1;
            break;

```

- fig05_08.c (1 of 3)

199

```c
         case 2: /* rolled 2 */
            ++frequency2;
            break;

         case 3: /* rolled 3 */
            ++frequency3;
            break;

         case 4: /* rolled 4 */
            ++frequency4;
            break;

         case 5: /* rolled 5 */
            ++frequency5;
            break;

         case 6: /* rolled 6 */
            ++frequency6;
            break; /* optional */
      } /* end switch */

   } /* end for */

```

- fig05_08.c (2 of 3)

200

```c
   /* display results in tabular format */
   printf( "%s%13s\n", "Face", "Frequency" );
   printf( "   1%13d\n", frequency1 );
   printf( "   2%13d\n", frequency2 );
   printf( "   3%13d\n", frequency3 );
   printf( "   4%13d\n", frequency4 );
   printf( "   5%13d\n", frequency5 );
   printf( "   6%13d\n", frequency6 );

   return 0; /* indicates successful termination */

} /* end main */
```

```
Face    Frequency
   1         1003
   2         1017
   3          983
   4          994
   5         1004
   6          999
```

- fig05_08.c (3 of 3)

Random Number Generation

- `srand` function
  - `<stdlib.h>`
  - Takes an integer seed and jumps to that location in its "random" sequence

    `srand(` *seed* **`);`**
  - `srand( time( NULL ) );/*load <time.h> */`
    - `time( NULL )`
      - Returns the time at which the program was compiled in seconds
      - “Randomizes" the seed

```c
/* Fig. 5.9: fig05_09.c
   Randomizing die-rolling program */
#include <stdlib.h>
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int i;         /* counter */
   unsigned seed; /* number used to seed random number generator */

   printf( "Enter seed: " );
   scanf( "%u", &seed ); /* note %u for unsigned */

   srand( seed ); /* seed random number generator */

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

```

Seeds the **rand** function

- fig05_09.c (1 of 2)

203

```c
      /* pick a random number from 1 to 6 and output it */
      printf( "%10d", 1 + ( rand() % 6 ) );

      /* if counter is divisible by 5, begin a new line of output */
      if ( i % 5 == 0 ) {
         printf( "\n" );
      } /* end if */

   } /* end for */

   return 0; /* indicates successful termination */

} /* end main */
```

```
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
```

```
Enter seed: 867
         2         4         6         1         6
         1         1         3         6         2
```

```
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
```

- fig05_09.c (2 of 2)

204

Example: A Game of Chance

- Craps simulator
- Rules
  - Roll two dice
    - 7 or 11 on first throw, player wins
    - 2, 3, or 12 on first throw, player loses
    - 4, 5, 6, 8, 9, 10 - value becomes player's "point"
  - Player must roll his point before rolling 7 to win

205

```c
/* Fig. 5.10: fig05_10.c
   Craps */
#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* contains prototype for function time */

/* enumeration constants represent game status */
enum Status { CONTINUE, WON, LOST };

int rollDice( void ); /* function prototype */

/* function main begins program execution */
int main( void )
{
   int sum;        /* sum of rolled dice */
   int myPoint;    /* point earned */

   enum Status gameStatus; /* can contain CONTINUE, WON, or LOST */

   /* randomize random number generator using current time */
   srand( time( NULL ) );

   sum = rollDice(); /* first roll of the dice */

   /* determine game status based on sum of dice */
   switch( sum ) {

```

- fig05_10.c (1 of 4)

enum (enumeration) assigns numerical values to **CONTINUE**, **WON** and **LOST**

206

```c
      /* win on first roll */
      case 7:
      case 11:
         gameStatus = WON;
         break;

      /* lose on first roll */
      case 2:
      case 3:
      case 12:
         gameStatus = LOST;
         break;

      /* remember point */
      default:
         gameStatus = CONTINUE;
         myPoint = sum;
         printf( "Point is %d\n", myPoint );
         break; /* optional */
   } /* end switch */

```

- fig05_10.c (2 of 4)

207

```c
   /* while game not complete */
   while ( gameStatus == CONTINUE ) {
      sum = rollDice(); /* roll dice again */

      /* determine game status */
      if ( sum == myPoint ) { /* win by making point */
         gameStatus = WON; /* game over, player won */
      } /* end if */
      else {

         if ( sum == 7 ) { /* lose by rolling 7 */
            gameStatus = LOST; /* game over, player lost */
         } /* end if */

      } /* end else */

   } /* end while */

   /* display won or lost message */
   if ( gameStatus == WON ) { /* did player win? */
      printf( "Player wins\n" );
   } /* end if */
   else { /* player lost */
      printf( "Player loses\n" );
   } /* end else */

   return 0; /* indicates successful termination */

} /* end main */
```

- fig05_10.c (3 of 4)

208

```c

/* roll dice, calculate sum and display results */
int rollDice( void )
{
   int die1;    /* first die */
   int die2;    /* second die */
   int workSum; /* sum of dice */

   die1 = 1 + ( rand() % 6 ); /* pick random die1 value */
   die2 = 1 + ( rand() % 6 ); /* pick random die2 value */
   workSum = die1 + die2;     /* sum die1 and die2 */

   /* display results of this roll */
   printf( "Player rolled %d + %d = %d\n", die1, die2, workSum );

   return workSum; /* return sum of dice */

} /* end function rollRice */
```

- fig05_10.c (4 of 4)

209

```
Player rolled 5 + 6 = 11
Player wins
```

```
Player rolled 4 + 1 = 5
Point is 5
Player rolled 6 + 2 = 8
Player rolled 2 + 1 = 3
Player rolled 3 + 2 = 5
Player wins
```

```
Player rolled 1 + 1 = 2
Player loses
```

```
Player rolled 6 + 4 = 10
Point is 10
Player rolled 3 + 4 = 7
Player loses
```

- fig05_11.c

210

Storage Classes

- Storage class specifiers
  - Storage duration – how long an object exists in memory
  - Scope – where object can be referenced in program
  - Linkage – specifies the files in which an identifier is known (more in Chapter 14)
- Automatic storage
  - Object created and destroyed within its block
  - `auto:` default for local variables

    ```
    auto double x, y;
    ```
  - `register:` tries to put variable into high-speed registers
    - Can only be used for automatic variables

      ```
      register int counter = 1;
      ```

211

Storage Classes

- Static storage
  - Variables exist for entire program execution
  - Default value of zero
  - `static:` local variables defined in functions.
    - Keep value after function ends
    - Only known in their own function
  - `extern:` default for global variables and functions
    - Known in any function

212

Scope Rules

- File scope
  - Identifier defined outside function, known in all functions
  - Used for global variables, function definitions, function prototypes
- Function scope
  - Can only be referenced inside a function body
  - Used only for labels (`start:, case:` , etc.)

213

Scope Rules

- Block scope
  - Identifier declared inside a block
    - Block scope begins at definition, ends at right brace
  - Used for variables, function parameters (local variables of function)
  - Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block
- Function prototype scope
  - Used for identifiers in parameter list

214

```c
/* Fig. 5.12: fig05_12.c
   A scoping example */
#include <stdio.h>

void useLocal( void );       /* function prototype */
void useStaticLocal( void ); /* function prototype */
void useGlobal( void );      /* function prototype */

int x = 1; /* global variable */

/* function main begins program execution */
int main( void )
{
   int x = 5; /* local variable to main */

   printf("local x in outer scope of main is %d\n", x );

   { /* start new scope */
      int x = 7; /* local variable to new scope */

      printf( "local x in inner scope of main is %d\n", x );
   } /* end new scope */

```

Global variable with file scope

Variable with block scope

Variable with block scope

- fig05_12.c (1 of 4)

215

```c
   printf( "local x in outer scope of main is %d\n", x );

   useLocal();       /* useLocal has automatic local x */
   useStaticLocal(); /* useStaticLocal has static local x */
   useGlobal();      /* useGlobal uses global x */
   useLocal();       /* useLocal reinitializes automatic local x */
   useStaticLocal(); /* static local x retains its prior value */
   useGlobal();      /* global x also retains its value */

   printf( "\nlocal x in main is %d\n", x );

   return 0; /* indicates successful termination */

} /* end main */

/* useLocal reinitializes local variable x during each call */
void useLocal( void )
{
   int x = 25;  /* initialized each time useLocal is called */

   printf( "\nlocal x in useLocal is %d after entering useLocal\n", x );
   x++;
   printf( "local x in useLocal is %d before exiting useLocal\n", x );
} /* end function useLocal */

```

Variable with block scope

- fig05_12.c (2 of 4)

216

```c
/* useStaticLocal initializes static local variable x only the first time
   the function is called; value of x is saved between calls to this
   function */
void useStaticLocal( void )
{
   /* initialized only first time useStaticLocal is called */
   static int x = 50;

   printf( "\nlocal static x is %d on entering useStaticLocal\n", x );
   x++;
   printf( "local static x is %d on exiting useStaticLocal\n", x );
} /* end function useStaticLocal */

/* function useGlobal modifies global variable x during each call */
void useGlobal( void )
{
   printf( "\nglobal x is %d on entering useGlobal\n", x );
   x *= 10;
   printf( "global x is %d on exiting useGlobal\n", x );
} /* end function useGlobal */
```

Static variable with block scope

Global variable

- fig05_12.c (3 of 4)

217

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

- fig05_12.c (4 of 4)

218

OBJECTIVES

- To construct programs modularly from small pieces called functions.
- The common math functions available in the C Standard Library.
- How to create new functions.
- The mechanisms used to pass information between functions.
- ★ How to write and use recursive functions, i.e., functions that call themselves.
- Review

219

Recursion

- Recursive functions
  - Functions that call themselves
  - Can only solve a base case
  - Divide a problem up into
    - What it can do
    - What it cannot do
      - What it cannot do resembles original problem
      - The function launches a new copy of itself (recursion step) to solve what it cannot do
  - Eventually base case gets solved
    - Gets plugged in, works its way up and solves whole problem

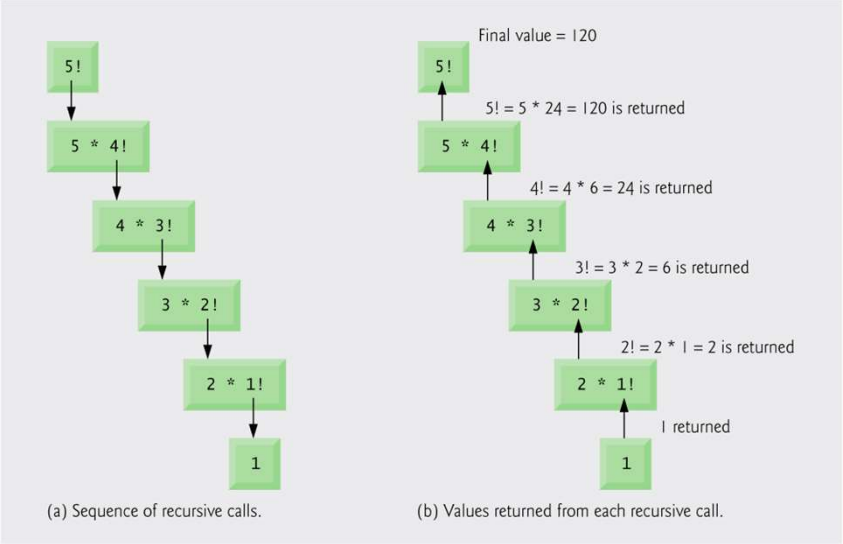
220

Recursion

- Example: factorials
  - `5! = 5 * 4 * 3 * 2 * 1`
  - Notice that
    - `5! = 5 * 4!`
    - `4! = 4 * 3! ...`
  - Can compute factorials recursively
  - Solve base case (`1! = 0! = 1`) then plug in
    - `2! = 2 * 1! = 2 * 1 = 2;`
    - `3! = 3 * 2! = 3 * 2 = 6;`

221

Recursive evaluation of 5!



(a) Sequence of recursive calls.

(b) Values returned from each recursive call.

222

```c
/* Fig. 5.14: fig05_14.c
   Recursive factorial function */
#include <stdio.h>

long factorial( long number ); /* function prototype */

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

   /* loop 11 times; during each iteration, calculate
      factorial( i ) and display result */
   for ( i = 0; i <= 10; i++ ) {
      printf( "%2d! = %ld\n", i, factorial( i ) );
   } /* end for */

   return 0; /* indicates successful termination */

} /* end main */

```

- fig05_14.c (1 of 2)

223

```c
/* recursive definition of function factorial */
long factorial( long number )
{
   /* base case */
   if ( number <= 1 ) {
      return 1;
   } /* end if */
   else { /* recursive step */
      return ( number * factorial( number - 1 ) );
   } /* end else */

} /* end function factorial */
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

- fig05_14.c (2 of 2)

224

Example Using Recursion: The Fibonacci Series

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - Each number is the sum of the previous two
  - Can be solved recursively:
    - `fib( n ) = fib( n - 1 ) + fib( n - 2 )`
  - Code for the `fibonacci` function

```
long fibonacci( long n )
{
  if (n == 0 || n == 1)  // base case
    return n;
  else
    return fibonacci( n - 1) +
       fibonacci( n - 2 );
}
```

225

```c
/* Fig. 5.15: fig05_15.c
   Recursive fibonacci function */
#include <stdio.h>

long fibonacci( long n ); /* function prototype */

/* function main begins program execution */
int main( void )
{
   long result; /* fibonacci value */
   long number; /* number input by user */

   /* obtain integer from user */
   printf( "Enter an integer: " );
   scanf( "%ld", &number );

   /* calculate fibonacci value for number input by user */
   result = fibonacci( number );

   /* display result */
   printf( "Fibonacci( %ld ) = %ld\n", number, result );

   return 0; /* indicates successful termination */

} /* end main */
```

- fig05_15.c (1 of 4)

226

```c
/* Recursive definition of function fibonacci */
long fibonacci( long n )
{
   /* base case */
   if ( n == 0 || n == 1 ) {
      return n;
   } /* end if */
   else { /* recursive step */
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
   } /* end else */

} /* end function fibonacci */
```

```
Enter an integer: 0
Fibonacci( 0 ) = 0
```

```
Enter an integer: 1
Fibonacci( 1 ) = 1
```

```
Enter an integer: 2
Fibonacci( 2 ) = 1
```

(continued on next slide…)

- fig05_15.c (2 of 4)

227

(continued from previous slide…)

```
Enter an integer: 3
Fibonacci( 3 ) = 2
```

```
Enter an integer: 4
Fibonacci( 4 ) = 3
```

```
Enter an integer: 5
Fibonacci( 5 ) = 5
```

```
Enter an integer: 6
Fibonacci( 6 ) = 8
```

(continued on next slide…)

- fig05_15.c (3 of 4)

228

(continued from previous slide…)

```
Enter an integer: 10
Fibonacci( 10 ) = 55
```

```
Enter an integer: 20
Fibonacci( 20 ) = 6765
```

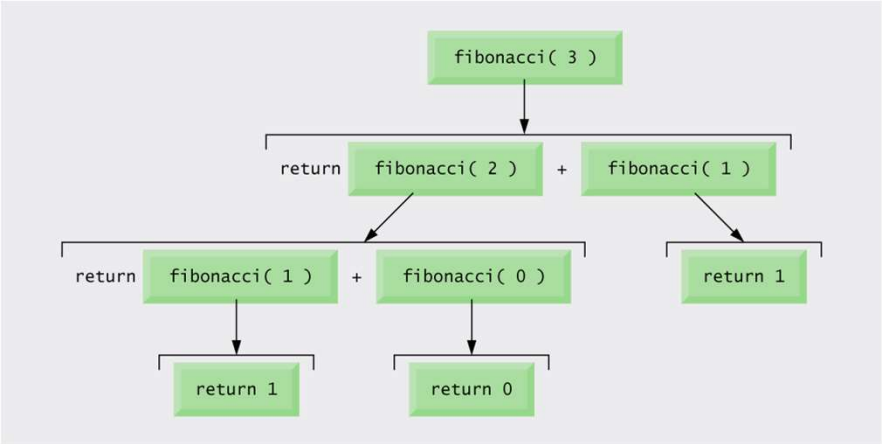
```
Enter an integer: 30
Fibonacci( 30 ) = 832040
```

```
Enter an integer: 35
Fibonacci( 35 ) = 9227465
```

- fig05_15.c (4 of 4)

229

Set of recursive calls for `fibonacci(3)`



230

Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance
  - Choice between performance (iteration) and good software engineering (recursion)

231

Recursion examples and exercises in the text (1)

Chapter	Recursion examples and exercises
Chapter 5	Factorial function Fibonacci function Greatest common divisor Sum of two integers Multiply two integers Raising an integer to an integer power Towers of Hanoi Recursive `main` Printing keyboard inputs in reverse Visualizing recursion
Chapter 6	Sum the elements of an array Print an array Print an array backward Print a string backward Check if a string is a palindrome Minimum value in an array Selection sort Quicksort Linear search Binary search

232

Recursion examples and exercises in the text (2)

Chapter	Recursion examples and exercises
Chapter 7	Eight Queens Maze traversal
Chapter 8	Printing a string input at the keyboard backward
Chapter 12	Linked list insert Linked list delete Search a linked list Print a linked list backward Binary tree insert Preorder traversal of a binary tree Inorder traversal of a binary tree Postorder traversal of a binary tree
Chapter 16	Selection sort Quicksort

233

OBJECTIVES

- To construct programs modularly from small pieces called functions.
- The common math functions available in the C Standard Library.
- How to create new functions.
- The mechanisms used to pass information between functions.
- How to write and use recursive functions, i.e., functions that call themselves.
- Review

234

Review

- Function is the basic module in C program.
- Math library functions.
- ★ Function definitions.
- Function prototypes.
- Header file.
- Calling functions: call-by-value and call-by-reference.
- ★ Random number generation.
- Storage classes.
- Scope rules.
- ★ Recursion: Fibonacci Series.

235

The End

- Thank you very much!

236

Chapter 6

C Arrays

Arrays are data structures consisting of related data items of the same type. Arrays are static entities in that they remain the same size throughout program execution.

Acknowledgment

The notes are adapted from those provided by Deitel & Associates, Inc. and Pearson Education Inc.

237

OBJECTIVES

- To define an array, initialize an array and refer to individual elements of an array.
- To define symbolic constants.
- To pass arrays to functions.
- To use arrays to store, sort and search lists and tables of values.
- To define and manipulate multiple-subscripted arrays.
- Review

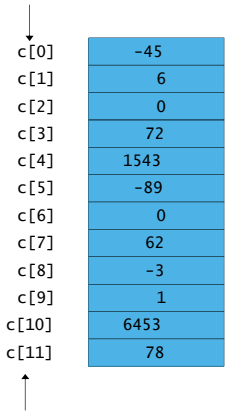
238

Arrays

- Array
  - Group of consecutive memory locations
  - Same name and type
- To refer to an element, specify
  - Array name
  - Position number
- Format:

  arrayname [*position number*]
  - First element at position 0
  - n element array named c:
    - c[0], c[1]...c[n – 1]

Name of array (Note that all elements of this array have the same name, c)



Position number of the element within array c

239

Arrays

- Array elements are like normal variables

```
c[ 0 ] =  3;
printf( "%d", c[ 0 ] );
```

  - Perform operations in subscript. If x equals 3

```
c[ 5 - 2 ] == c[ 3 ] == c[ x ]
```

- Notice!!
  - "seventh element of the array" → the array element with the subscript 6
  - "array element seven" → the array element with the subscript of 7, i.e. the 8th element in the array

240

Operator Precedence

Operators	Associativity	Type
`[]` `()`	left to right	highest
`++` `--` `!` (*type*)	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

241

Defining Arrays

- When defining arrays, specify
  - Name
  - Type of array
  - Number of elements
    `arrayType arrayName[ numberOfElements ];`
  - Examples:
    ```
    int c[ 10 ];
    float myArray[ 3284 ];
    ```
- Defining multiple arrays of same type
  - Format similar to regular variables
  - Example:
    `int b[ 100 ], x[ 27 ];`

242

Array Examples

- Initializers

  `int n[ 5 ] = { 1, 2, 3, 4, 5 };`

  - If not enough initializers, rightmost elements become 0

    `int n[ 5 ] = { 0 }`

    - All elements 0
  - If too many initializers, a syntax error occurs
  - C arrays have no bounds checking
- If size omitted, initializers determine it

  `int n[ ] = { 1, 2, 3, 4, 5 };`

  - 5 initializers, therefore 5 element array

243

```c
/* Fig. 6.3: fig06_03.c
   initializing an array */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int n[ 10 ]; /* n is an array of 10 integers */
   int i; /* counter */

   /* initialize elements of array n to 0 */
   for ( i = 0; i < 10; i++ ) {
      n[ i ] = 0; /* set element at location i to 0 */
   } /* end for */

   printf( "%s%13s\n", "Element", "Value" );

   /* output contents of array n in tabular format */
   for ( i = 0; i < 10; i++ ) {
      printf( "%7d%13d\n", i, n[ i ] );
   } /* end for */

   return 0; /* indicates successful termination */

} /* end main */
```

- fig06_03.c (1 of 2)

for loop initializes each array element separately

for loop outputs all array elements

244

```
Element       Value
      0           0
      1           0
      2           0
      3           0
      4           0
      5           0
      6           0
      7           0
      8           0
      9           0
```

- fig06_03.c (2 of 2)

245

```c
/* Fig. 6.4: fig06_04.c
   Initializing an array with an initializer list */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   /* use initializer list to initialize array n */
   int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
   int i; /* counter */

   printf( "%s%13s\n", "Element", "Value" );

   /* output contents of array in tabular format */
   for ( i = 0; i < 10; i++ ) {
      printf( "%7d%13d\n", i, n[ i ] );
   } /* end for */

   return 0; /* indicates successful termination */

} /* end main */
```

- fig06_04.c (1 of 2)

initializer list initializes all array elements simultaneously

246

```
Element        Value
      0           32
      1           27
      2           64
      3           18
      4           95
      5           14
      6           90
      7           70
      8           60
      9           37
```

- fig06_04.c
  (2 of 2)

247

OBJECTIVES

- To define an array, initialize an array and refer to individual elements of an array.
- To define symbolic constants.
- To pass arrays to functions.
- To use arrays to store, sort and search lists and tables of values.
- To define and manipulate multiple-subscripted arrays.
- Review

248

```c
/* Fig. 6.5: fig06_05.c
   Initialize the elements of array s to the even integers from 2 to 20 */
#include <stdio.h>
#define SIZE 10 /* maximum size of array */

/* function main begins program execution */
int main( void )
{
   /* symbolic constant SIZE can be used to specify array size */
   int s[ SIZE ]; /* array s has SIZE elements */
   int j; /* counter */

   for ( j = 0; j < SIZE; j++ ) { /* set the values */
      s[ j ] = 2 + 2 * j;
   } /* end for */

   printf( "%s%13s\n", "Element", "Value" );

   /* output contents of array s in tabular format */
   for ( j = 0; j < SIZE; j++ ) {
      printf( "%7d%13d\n", j, s[ j ] );
   } /* end for */

   return 0; /* indicates successful termination */

} /* end main */
```

`#define` directive tells compiler to replace all instances of the word **`SIZE`** with **`10`**

`SIZE` is replaced with **`10`** by the compiler, so array **`s`** has 10 elements

`for` loop initializes each array element separately

- fig06_05.c (1 of 2)

249

```
Element        Value
      0            2
      1            4
      2            6
      3            8
      4           10
      5           12
      6           14
      7           16
      8           18
      9           20
```

- fig06_05.c (2 of 2)

250

```c
/* Fig. 6.6: fig06_06.c
   Compute the sum of the elements of the array */
#include <stdio.h>
#define SIZE 12

/* function main begins program execution */
int main( void )
{
   /* use initializer list to initialize array */
   int a[ SIZE ] = { 1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45 };
   int i; /* counter */
   int total = 0; /* sum of array */

   /* sum contents of array a */
   for ( i = 0; i < SIZE; i++ ) {
      total += a[ i ];
   } /* end for */

   printf( "Total of array element values is %d\n", total );

   return 0; /* indicates successful termination */

} /* end main */
```

```
Total of array element values is 383
```

- fig06_06.c

initializer list initializes all array elements simultaneously

for loop adds each element of the array to variable **total**

251

```c
/* Fig. 6.7: fig06_07.c
   Student poll program */
#include <stdio.h>
#define RESPONSE_SIZE 40 /* define array sizes */
#define FREQUENCY_SIZE 11

/* function main begins program execution */
int main( void )
{
   int answer; /* counter to loop through 40 responses */
   int rating; /* counter to loop through frequencies 1-10 */

   /* initialize frequency counters to 0 */
   int frequency[ FREQUENCY_SIZE ] = { 0 };

   /* place the survey responses in the responses array */
   int responses[ RESPONSE_SIZE ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
        1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
        5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };

   /* for each answer, select value of an element of array responses
        and use that value as subscript in array frequency to
        determine element to increment */
   for ( answer = 0; answer < RESPONSE_SIZE; answer++ ) {
      ++frequency[ responses [ answer ] ];
   } /* end for */

```

- fig06_07.c (1 of 2)

#define directives create symbolic constants

frequency array is defined with 11 elements

responses array is defined with 40 elements and its elements are initialized

subscript of **frequency** array is given by value in **responses** array

252

```c
   /* display results */
   printf( "%s%17s\n", "Rating", "Frequency" );

   /* output the frequencies in a tabular format */
   for ( rating = 1; rating < FREQUENCY_SIZE; rating++ ) {
      printf( "%6d%17d\n", rating, frequency[ rating ] );
   } /* end for */

   return 0; /* indicates successful termination */

} /* end main */
```

```
Rating        Frequency
     1                2
     2                2
     3                2
     4                2
     5                5
     6               11
     7                5
     8                7
     9                1
    10                3
```

- fig06_07.c (2 of 2)

253

```c
/* Fig. 6.8: fig06_08.c
   Histogram printing program */
#include <stdio.h>
#define SIZE 10

/* function main begins program execution */
int main( void )
{
   /* use initializer list to initialize array n */
   int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
   int i; /* outer for counter for array elements */
   int j; /* inner for counter counts *s in each histogram bar */

   printf( "%s%13s%17s\n", "Element", "Value", "Histogram" );

   /* for each element of array n, output a bar of the histogram */
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%7d%13d        ", i, n[ i ]) ;

      for ( j = 1; j <= n[ i ]; j++ ) { /* print one bar */
         printf( "%c", '*' );
       } /* end inner for */

      printf( "\n" ); /* end a histogram bar */
   } /* end outer for */

   return 0; /* indicates successful termination */

} /* end main */
```

nested **for** loop prints n[**i**] asterisks on the **i**th line

- fig06_08.c (1 of 2)

254

```
Element        Value        Histogram
      0           19        *******************
      1            3        ***
      2           15        ***************
      3            7        *******
      4           11        ***********
      5            9        *********
      6           13        *************
      7            5        *****
      8           17        *****************
      9            1        *
```

- fig06_08.c (2 of 2)

255

```
1  /* Fig. 6.9: fig06_09.c
2     Roll a six-sided die 6000 times */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6  #define SIZE 7
7
8  /* function main begins program execution */
9  int main( void )
10 {
11    int face; /* random die value 1 - 6 */
12    int roll; /* roll counter 1-6000 */
13    int frequency[ SIZE ] = { 0 }; /* clear counts */
14
15    srand( time( NULL ) ); /* seed random-number generator */
16
17    /* roll die 6000 times */
18    for ( roll = 1; roll <= 6000; roll++ ) {
19       face = 1 + rand() % 6;
20       ++frequency[ face ]; /* replaces 26-line switch of Fig. 5.8 */
21    } /* end for */
```

- fig06_09.c (1 of 2)

for loop uses one array to track number of times each number is rolled instead of using 6 variables and a **switch** statement

256

```c

   printf( "%s%17s\n", "Face", "Frequency" );

   /* output frequency elements 1-6 in tabular format */
   for ( face = 1; face < SIZE; face++ ) {
      printf( "%4d%17d\n", face, frequency[ face ] );
   } /* end for */

   return 0; /* indicates successful termination */

} /* end main */
```

```
Face        Frequency
   1             1029
   2              951
   3              987
   4             1033
   5             1010
   6              990
```

- fig06_09.c (2 of 2)

257

Array Examples

- Character arrays
  - String "`first`" is really a static array of characters
  - Character arrays can be initialized using string literals
    `char string1[] = "first";`
    - Null character `'\0'` terminates strings
    - `string1` actually has 6 elements
      - It is equivalent to

    `char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };`
  - Can access individual characters
    `string1[ 3 ] is character 's'`
  - Array name is address of array, so & not needed for scanf
    `scanf( "%s", string2 );`
    - Reads characters until whitespace encountered
    - Can write beyond end of array, be careful

258

```c
/* Fig. 6.10: fig06_10.c
   Treating character arrays as strings */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   char string1[ 20 ]; /* reserves 20 characters */
   char string2[] = "string literal"; /* reserves 15 characters */
   int i; /* counter */

   /* read string from user into array string1 */
   printf("Enter a string: ");
   scanf( "%s", string1 ); /* input ended by whitespace character */

   /* output strings */
   printf( "string1 is: %s\nstring2 is: %s\n"
           "string1 with spaces between characters is:\n",
           string1, string2 );

   /* output characters until null character is reached */
   for ( i = 0; string1[ i ] != '\0'; i++ ) {
      printf( "%c ", string1[ i ] );
   } /* end for */

   printf( "\n" );

   return 0; /* indicates successful termination */

} /* end main */
```

string2 array is defined with one element for each character, so 15 elements including null character **/0**

for loop prints characters of **string1** array with spaces in between

- fig06_10.c (1 of 2)

259

```
Enter a string: Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
```

- fig06_10.c (2 of 2)

260

Notice!

- In functions that contain automatic arrays where the function is in and out of scope frequently, make the array `static` so it is not created each time the function is called.

261

```c
/* Fig. 6.11: fig06_11.c
   Static arrays are initialized to zero */
#include <stdio.h>

void staticArrayInit( void );    /* function prototype */
void automaticArrayInit( void ); /* function prototype */

/* function main begins program execution */
int main( void )
{
   printf( "First call to each function:\n" );
   staticArrayInit();
   automaticArrayInit();

   printf( "\n\nSecond call to each function:\n" );
   staticArrayInit();
   automaticArrayInit();

   return 0; /* indicates successful termination */

} /* end main */

```

- fig06_11.c (1 of 4)

262

```c
/* function to demonstrate a static local array */
void staticArrayInit( void )
{
   /* initializes elements to 0 first time function is called */
   static int array1[ 3 ];
   int i; /* counter */

   printf( "\nValues on entering staticArrayInit:\n" );

   /* output contents of array1 */
   for ( i = 0; i <= 2; i++ ) {
      printf( "array1[ %d ] = %d  ", i, array1[ i ] );
   } /* end for */

   printf( "\nValues on exiting staticArrayInit:\n" );

   /* modify and output contents of array1 */
   for ( i = 0; i <= 2; i++ ) {
      printf( "array1[ %d ] = %d  ", i, array1[ i ] += 5 );
   } /* end for */

} /* end function staticArrayInit */
```

static array is created only once, when **staticArrayInit** is first called

- fig06_11.c (2 of 4)

263

```c

/* function to demonstrate an automatic local array */
void automaticArrayInit( void )
{
   /* initializes elements each time function is called */
   int array2[ 3 ] = { 1, 2, 3 };
   int i; /* counter */

   printf( "\n\nValues on entering automaticArrayInit:\n" );

   /* output contents of array2 */
   for ( i = 0; i <= 2; i++ ) {
      printf("array2[ %d ] = %d  ", i, array2[ i ] );
   } /* end for */

   printf( "\nValues on exiting automaticArrayInit:\n" );

   /* modify and output contents of array2 */
   for ( i = 0; i <= 2; i++ ) {
      printf( "array2[ %d ] = %d  ", i, array2[ i ] += 5 );
   } /* end for */

} /* end function automaticArrayInit */
```

automatic array is recreated every time **automaticArrayInit** is called

- fig06_11.c (3 of 4)

264

```
First call to each function:

Values on entering staticArrayInit:
array1[ 0 ] = 0  array1[ 1 ] = 0  array1[ 2 ] = 0
Values on exiting staticArrayInit:
array1[ 0 ] = 5  array1[ 1 ] = 5  array1[ 2 ] = 5

Values on entering automaticArrayInit:
array2[ 0 ] = 1  array2[ 1 ] = 2  array2[ 2 ] = 3
Values on exiting automaticArrayInit:
array2[ 0 ] = 6  array2[ 1 ] = 7  array2[ 2 ] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[ 0 ] = 5  array1[ 1 ] = 5  array1[ 2 ] = 5
Values on exiting staticArrayInit:
array1[ 0 ] = 10  array1[ 1 ] = 10  array1[ 2 ] = 10

Values on entering automaticArrayInit:
array2[ 0 ] = 1  array2[ 1 ] = 2  array2[ 2 ] = 3
Values on exiting automaticArrayInit:
array2[ 0 ] = 6  array2[ 1 ] = 7  array2[ 2 ] = 8
```

- fig06_11.c (4 of 4)

265

OBJECTIVES

- To define an array, initialize an array and refer to individual elements of an array.
- To define symbolic constants.
- To pass arrays to functions.
- To use arrays to store, sort and search lists and tables of values.
- To define and manipulate multiple-subscripted arrays.
- Review

266

Passing Arrays to Functions

- Passing arrays
  - To pass an array argument to a function, specify the name of the array without any brackets

```
int myArray[ 24 ];
myFunction( myArray, 24 );
```

    - Array size usually passed to function
  - Arrays passed call-by-reference
  - Name of array is address of first element
  - Function knows where the array is stored
    - Modifies original memory locations
- Passing array elements
  - Passed by call-by-value
  - Pass subscripted name (i.e., `myArray[ 3 ]`) to function

267

Passing Arrays to Functions

- Function prototype

```
void modifyArray( int b[], int arraySize );
```

  - Parameter names optional in prototype
    - `int b[]` could be written `int []`
    - `int arraySize` could be simply `int`

268

```c
/* Fig. 6.12: fig06_12.c
   The name of an array is the same as &array[ 0 ] */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   char array[ 5 ]; /* define an array of size 5 */

   printf( "    array = %p\n&array[0] = %p\n   &array = %p\n",
      array, &array[ 0 ], &array );

   return 0; /* indicates successful termination */

} /* end main */
```

```
    array = 0012FF78
&array[0] = 0012FF78
   &array = 0012FF78
```

- fig06_12.c

Note: '%p' conversion specifier normally outputs addresses as hexadecimal numbers.

269

```c
/* Fig. 6.13: fig06_13.c
   Passing arrays and individual array elements to functions */
#include <stdio.h>
#define SIZE 5

/* function prototypes */
void modifyArray( int b[], int size );
void modifyElement( int e );

/* function main begins program execution */
int main( void )
{
   int a[ SIZE ] = { 0, 1, 2, 3, 4 }; /* initialize a */
   int i; /* counter */

   printf( "Effects of passing entire array by reference:\n\nThe "
         "values of the original array are:\n" );

   /* output original array */
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%3d", a[ i ] );
   } /* end for */

   printf( "\n" );

   /* pass array a to modifyArray by reference */
   modifyArray( a, SIZE );

   printf( "The values of the modified array are:\n" );
```

Function prototype indicates function will take an array

Array **a** is passed to **modifyArray** by passing only its name

- fig06_13.c (1 of 3)

270

```c
   /* output modified array */
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%3d", a[ i ] );
   } /* end for */

   /* output value of a[ 3 ] */
   printf( "\n\n\nEffects of passing array element "
           "by value:\n\nThe value of a[3] is %d\n", a[ 3 ] );

   modifyElement( a[ 3 ] ); /* pass array element a[ 3 ] by value */

   /* output value of a[ 3 ] */
   printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );

   return 0; /* indicates successful termination */

} /* end main */

/* in function modifyArray, "b" points to the original array "a"
   in memory */
void modifyArray( int b[], int size )
{
   int j; /* counter */

   /* multiply each array element by 2 */
   for ( j = 0; j < size; j++ ) {
      b[ j ] *= 2;
   } /* end for */

} /* end function modifyArray */
```

Array element is passed to **modifyElement** by passing **a[3]**

- fig06_13.c (2 of 3)

271

```c

/* in function modifyElement, "e" is a local copy of array element
   a[ 3 ] passed from main */
void modifyElement( int e )
{
   /* multiply parameter by 2 */
   printf( "Value in modifyElement is %d\n", e *= 2 );
} /* end function modifyElement */
```

```
Effects of passing entire array by reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8


Effects of passing array element by value:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[ 3 ] is 6
```

- fig06_13.c (3 of 3)

272

```c
/* Fig. 6.14: fig06_14.c
   Demonstrating the const type qualifier with arrays */
#include <stdio.h>

void tryToModifyArray( const int b[] ); /* function prototype */

/* function main begins program execution */
int main( void )
{
   int a[] = { 10, 20, 30 }; /* initialize a */

   tryToModifyArray( a );

   printf("%d %d %d\n", a[ 0 ], a[ 1 ], a[ 2 ] );

   return 0; /* indicates successful termination */

} /* end main */

/* in function tryToModifyArray, array b is const, so it cannot be
   used to modify the original array a in main. */
void tryToModifyArray( const int b[] )
{
   b[ 0 ] /= 2;    /* error */
   b[ 1 ] /= 2;    /* error */
   b[ 2 ] /= 2;    /* error */
} /* end function tryToModifyArray */
```

`const` qualifier tells compiler that array cannot be changed

Any attempts to modify the array will result in errors

- fig06_14.c (1 of 2)

273

```
Compiling...
FIG06_14.C
fig06_14.c(24) : error C2166: l-value specifies const object
fig06_14.c(25) : error C2166: l-value specifies const object
fig06_14.c(26) : error C2166: l-value specifies const object
```

- fig06_14.c (2 of 2)

274

OBJECTIVES

- To define an array, initialize an array and refer to individual elements of an array.
- To define symbolic constants.
- To pass arrays to functions.
- To use arrays to store, sort and search lists and tables of values.
- To define and manipulate multiple-subscripted arrays.
- Review

275

Sorting Arrays

- Sorting data
  - Important computing application
  - Virtually every organization must sort some data
- Bubble sort (sinking sort)
  - Several passes through the array
  - Successive pairs of elements are compared
    - If increasing order (or identical), no change
    - If decreasing order, elements exchanged
  - Repeat
- Example:
  - original: 3 4 2 6 7
  - pass 1: 3 2 4 6 7
  - pass 2: 2 3 4 6 7
  - Small elements "bubble" to the top

276

```c
/* Fig. 6.15: fig06_15.c
   This program sorts an array's values into ascending order */
#include <stdio.h>
#define SIZE 10

/* function main begins program execution */
int main( void )
{
   /* initialize a */
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
   int pass; /* passes counter */
   int i;    /* comparisons counter */
   int hold; /* temporary location used to swap array elements */

   printf( "Data items in original order\n" );

   /* output original array */
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%4d", a[ i ] );
   } /* end for */

   /* bubble sort */
   /* loop to control number of passes */
   for ( pass = 1; pass < SIZE; pass++ ) {

      /* loop to control number of comparisons per pass */
      for ( i = 0; i < SIZE - 1; i++ ) {

```

- fig06_15.c (1 of 2)

277

```c
         /* compare adjacent elements and swap them if first
         element is greater than second element */
         if ( a[ i ] > a[ i + 1 ] ) {
            hold = a[ i ];
            a[ i ] = a[ i + 1 ];
            a[ i + 1 ] = hold;
         } /* end if */

      } /* end inner for */

   } /* end outer for */

   printf( "\nData items in ascending order\n" );

   /* output sorted array */
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%4d", a[ i ] );
   } /* end for */

   printf( "\n" );

   return 0; /* indicates successful termination */
}
```

If any two array elements are out of order, the function swaps them

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

- fig06_15.c (2 of 2)

278

Case Study: Computing Mean, Median and Mode Using Arrays

- Mean – average
- Median – number in middle of sorted list
  - 1, 2, 3, 4, 5
  - 3 is the median
- Mode – number that occurs most often
  - 1, 1, 1, 2, 3, 3, 4, 5
  - 1 is the mode

279

```c
/* Fig. 6.16: fig06_16.c
   This program introduces the topic of survey data analysis.
   It computes the mean, median and mode of the data */
#include <stdio.h>
#define SIZE 99

/* function prototypes */
void mean( const int answer[] );
void median( int answer[] );
void mode( int freq[], const int answer[] ) ;
void bubbleSort( int a[] );
void printArray( const int a[] );

/* function main begins program execution */
int main( void )
{
   int frequency[ 10 ] = { 0 }; /* initialize array frequency */

   /* initialize array response */
   int response[ SIZE ] =
      { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
        7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
        6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
        7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
        6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
        7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
        5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
        7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
        7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
        4, 5, 6, 1, 6, 5, 7, 8, 7 };
```

- fig06_16.c (1 of 6)

280

```c

   /* process responses */
   mean( response );
   median( response );
   mode( frequency, response );

   return 0; /* indicates successful termination */

} /* end main */

/* calculate average of all response values */
void mean( const int answer[] )
{
   int j; /* counter for totaling array elements */
   int total = 0; /* variable to hold sum of array elements */

   printf( "%s\n%s\n%s\n", "********", "  Mean", "********" );

   /* total response values */
   for ( j = 0; j < SIZE; j++ ) {
      total += answer[ j ];
   } /* end for */

   printf( "The mean is the average value of the data\n"
           "items. The mean is equal to the total of\n"
           "all the data items divided by the number\n"
           "of data items ( %d ). The mean value for\n"
           "this run is: %d / %d = %.4f\n\n",
           SIZE, total, SIZE, (  double  ) total / SIZE );
} /* end function mean */
```

- fig06_16.c (2 of 6)

281

```c

/* sort array and determine median element's value */
void median( int answer[] )
{
   printf( "\n%s\n%s\n%s\n%s",
           "********", " Median", "********",
           "The unsorted array of responses is" );

   printArray( answer ); /* output unsorted array */

   bubbleSort( answer ); /* sort array */

   printf( "\n\nThe sorted array is" );
   printArray( answer ); /* output sorted array */

   /* display median element */
   printf( "\n\nThe median is element %d of\n"
           "the sorted %d element array.\n"
           "For this run the median is %d\n\n",
           SIZE / 2, SIZE, answer[ SIZE / 2 ] );
} /* end function median */

/* determine most frequent response */
void mode( int freq[], const int answer[] )
{
   int rating; /* counter for accessing elements 1-9 of array freq */
   int j; /* counter for summarizing elements 0-98 of array answer */
   int h; /* counter for diplaying histograms of elements in array freq */
   int largest = 0; /* represents largest frequency */
   int modeValue = 0; /* represents most frequent response */
```

Once the array is sorted, the median will be the value of the middle element

- fig06_16.c (3 of 6)

282

```c

   printf( "\n%s\n%s\n%s\n",
           "********", "  Mode", "********" );

   /* initialize frequencies to 0 */
   for ( rating = 1; rating <= 9; rating++ ) {
      freq[ rating ] = 0;
   } /* end for */

   /* summarize frequencies */
   for ( j = 0; j < SIZE; j++ ) {
      ++freq[ answer[ j ] ];
   } /* end for */

   /* output headers for result columns */
   printf( "%s%11s%19s\n\n%54s\n%54s\n\n",
           "Response", "Frequency", "Histogram",
           "1    1    2    2", "5    0    5    0    5" );

   /* output results */
   for ( rating = 1; rating <= 9; rating++ ) {
      printf( "%8d%11d          ", rating, freq[ rating ] );

      /* keep track of mode value and largest frequency value */
      if ( freq[ rating ] > largest ) {
         largest = freq[ rating ];
         modeValue = rating;
      } /* end if */
```

- fig06_16.c (4 of 6)

283

```c

      /* output histogram bar representing frequency value */
      for ( h = 1; h <= freq[ rating ]; h++ ) {
         printf( "*" );
      } /* end inner for */

      printf( "\n" ); /* being new line of output */
   } /* end outer for */

   /* display the mode value */
   printf( "The mode is the most frequent value.\n"
           "For this run the mode is %d which occurred"
           " %d times.\n", modeValue, largest );
 } /* end function mode */

/* function that sorts an array with bubble sort algorithm */
void bubbleSort( int a[] )
{
   int pass; /* pass counter */
   int j;    /* comparison counter */
   int hold; /* temporary location used to swap elements */

   /* loop to control number of passes */
   for ( pass = 1; pass < SIZE; pass++ ) {

      /* loop to control number of comparisons per pass */
      for ( j = 0; j < SIZE - 1; j++ ) {

```

- fig06_16.c (5 of 6)

284

```c
         /* swap elements if out of order */
         if ( a[ j ] > a[ j + 1 ] ) {
            hold = a[ j ];
            a[ j ] = a[ j + 1 ];
            a[ j + 1 ] = hold;
         } /* end if */

      } /* end inner for */

   } /* end outer for */

} /* end function bubbleSort */

/* output array contents (20 values per row) */
void printArray( const int a[] )
{
   int j; /* counter */

   /* output array contents */
   for ( j = 0; j < SIZE; j++ ) {

      if ( j % 20 == 0 ) { /* begin new line every 20 values */
         printf( "\n" );
      } /* end if */

      printf( "%2d", a[ j ] );
   } /* end for */

} /* end function printArray */
```

- fig06_16.c (6 of 6)

285

```
********
  Mean
********
The mean is the average value of the data
items. The mean is equal to the total of
all the data items divided by the number
of data items ( 99 ). The mean value for
this run is: 681 / 99 = 6.8788

********
 Median
********
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7

The sorted array is
 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5
 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
```

(continued on next slide…)

- (1 of 2)

286

(continued from previous slide…)

```
The median is element 49 of
the sorted 99 element array.
For this run the median is 7

********
  Mode
********
Response  Frequency          Histogram

                                     1    1    2    2
                                5    0    5    0    5

       1         1          *
       2         3          ***
       3         4          ****
       4         5          *****
       5         8          ********
       6         9          *********
       7        23          ***********************
       8        27          ***************************
       9        19          *******************
The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.
```

- (2 of 2)

287

Searching Arrays: Linear Search and Binary Search

- Search an array for a *key value*
- Linear search
  - Simple
  - Compare each element of array with key value
  - Useful for small and unsorted arrays
- Binary search
  - For sorted arrays
  - Compares `middle` element with `key`
    - If equal, match found
    - If `key < middle`, looks in first half of array
    - If `key > middle`, looks in last half
    - Repeat
  - Very fast; at most n steps, where 2^n > number of elements
    - 30 element array takes at most 5 steps
      - $2^5 > 30$ so at most 5 steps

288

```c
/* Fig. 6.18: fig06_18.c
   Linear search of an array */
#include <stdio.h>
#define SIZE 100

/* function prototype */
int linearSearch( const int array[], int key, int size );

/* function main begins program execution */
int main( void )
{
   int a[ SIZE ]; /* create array a */
   int x; /* counter for initializing elements 0-99 of array a */
   int searchKey; /* value to locate in array a */
   int element; /* variable to hold location of searchKey or -1 */

   /* create data */
   for ( x = 0; x < SIZE; x++ ) {
      a[ x ] = 2 * x;
   } /* end for */

```

- fig06_18.c (1 of 3)

289

```c
   printf( "Enter integer search key:\n" );
   scanf( "%d", &searchKey );

   /* attempt to locate searchKey in array a */
   element = linearSearch( a, searchKey, SIZE );

   /* display results */
   if ( element != -1 ) {
      printf( "Found value in element %d\n", element );
   } /* end if */
   else {
      printf( "Value not found\n" );
   } /* end else */

   return 0; /* indicates successful termination */

} /* end main */

/* compare key to every element of array until the location is found
   or until the end of array is reached; return subscript of element
   if key or -1 if key is not found */
int linearSearch( const int array[], int key, int size )
{
   int n; /* counter */

```

- fig06_18.c (2 of 3)

290

```c
   /* loop through array */
   for ( n = 0; n < size; ++n ) {

      if ( array[ n ] == key ) {
         return n; /* return location of key */
      } /* end if */

   } /* end for */

   return -1; /* key not found */

} /* end function linearSearch */
```

Linear search algorithm searches through every element in the array until a match is found

```
Enter integer search key:
36
Found value in element 18
```

```
Enter integer search key:
37
Value not found
```

- fig06_18.c (3 of 3)

291

```c
/* Fig. 6.19: fig06_19.c
   Binary search of an array */
#include <stdio.h>
#define SIZE 15

/* function prototypes */
int binarySearch( const int b[], int searchKey, int low, int high );
void printHeader( void );
void printRow( const int b[], int low, int mid, int high );

/* function main begins program execution */
int main( void )
{
   int a[ SIZE ]; /* create array a */
   int i; /* counter for initializing elements 0-14 of array a */
   int key; /* value to locate in array a */
   int result; /* variable to hold location of key or -1 */

   /* create data */
   for ( i = 0; i < SIZE; i++ ) {
      a[ i ] = 2 * i;
   } /* end for */

   printf( "Enter a number between 0 and 28: " );
   scanf( "%d", &key );

   printHeader();

   /* search for key in array a */
   result = binarySearch( a, key, 0, SIZE - 1 );
```

- fig06_19.c (1 of 6)

292

```c

   /* display results */
   if ( result != -1 ) {
      printf( "\n%d found in array element %d\n", key, result );
   } /* end if */
   else {
      printf( "\n%d not found\n", key );
   } /* end else */

   return 0; /* indicates successful termination */

} /* end main */

/* function to perform binary search of an array */
int binarySearch( const int b[], int searchKey, int low, int high )
{
   int middle; /* variable to hold middle element of array */

   /* loop until low subscript is greater than high subscript */
   while ( low <= high ) {

      /* determine middle element of subarray being searched */
      middle = ( low + high ) / 2;

      /* display subarray used in this loop iteration */
      printRow( b, low, middle, high );

```

- fig06_19.c (2 of 6)

293

```c
      /* if searchKey matched middle element, return middle */
      if ( searchKey == b[ middle ] ) {
         return middle;
      } /* end if */

      /* if searchKey less than middle element, set new high */
      else if ( searchKey < b[ middle ] ) {
         high = middle - 1; /* search low end of array */
      } /* end else if */

      /* if searchKey greater than middle element, set new low */
      else {
         low = middle + 1; /* search high end of array */
      } /* end else */

   } /* end while */

   return -1;   /* searchKey not found */

} /* end function binarySearch */

/* Print a header for the output */
void printHeader( void )
{
   int i; /* counter */

   printf( "\nSubscripts:\n" );

```

If value is found, return its index

If value is too high, search the left half of array

If value is too low, search the right half of array

- fig06_19.c (3 of 6)

294

```c
   /* output column head */
   for ( i = 0; i < SIZE; i++ ) {
      printf( "%3d ", i );
   } /* end for */

   printf( "\n" ); /* start new line of output */

   /* output line of - characters */
   for ( i = 1; i <= 4 * SIZE; i++ ) {
      printf( "-" );
   } /* end for */

   printf( "\n" ); /* start new line of output */
} /* end function printHeader */

/* Print one row of output showing the current
   part of the array being processed. */
void printRow( const int b[], int low, int mid, int high )
{
   int i; /* counter for iterating through array b */

```

- fig06_19.c (4 of 6)

295

```c
   /* loop through entire array */
   for ( i = 0; i < SIZE; i++ ) {

      /* display spaces if outside current subarray range */
      if ( i < low || i > high ) {
         printf( "    " );
      } /* end if */
      else if ( i == mid ) { /* display middle element */
         printf( "%3d*", b[ i ] ); /* mark middle value */
      } /* end else if */
      else { /* display other elements in subarray */
         printf( "%3d ", b[ i ] );
      } /* end else */

   } /* end for */

   printf( "\n" ); /* start new line of output */
} /* end function printRow */
```

```
Enter a number between 0 and 28: 25

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
---------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
                                 16  18  20  22* 24  26  28
                                                 24  26* 28
                                                 24*

25 not found
```

(continued on next slide…)

- fig06_19.c (5 of 6)

296

(continued from previous slide…)

```
Enter a number between 0 and 28: 8

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
-----------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12
                  8  10* 12
                  8*

8 found in array element 4
```

```
Enter a number between 0 and 28: 6

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
-----------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12

6 found in array element 3
```

- fig06_19.c (6 of 6)

297

OBJECTIVES

- To define an array, initialize an array and refer to individual elements of an array.
- To define symbolic constants.
- To pass arrays to functions.
- To use arrays to store, sort and search lists and tables of values.
- To define and manipulate multiple-subscripted arrays.
- Review

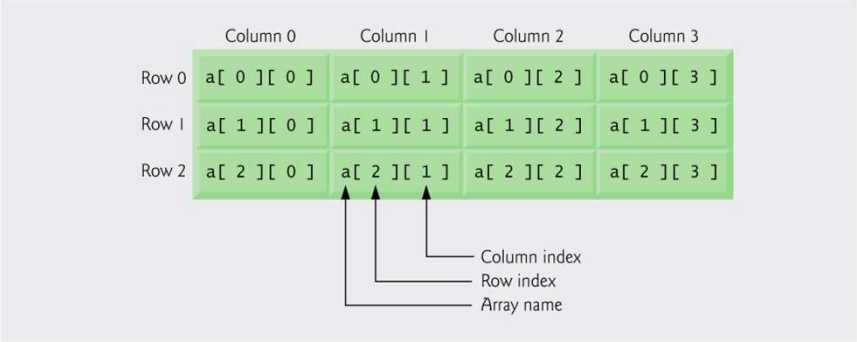
298

Multiple-Subscripted Arrays

- Multiple subscripted arrays
  - Tables with rows and columns (`m` by `n` array)
  - Like matrices: specify row, then column
- Initialization
  - `int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };`
  - Initializers grouped by row in braces
  - If not enough, unspecified elements set to zero
    `int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };`
- Referencing elements
  - Specify row, then column
    `printf( "%d", b[ 0 ][ 1 ] );`
- C interprets `a[ x, y ]` as `a[ y ]`

299

Double-subscripted array with three rows and four columns



300

```c
/* Fig. 6.21: fig06_21.c
   Initializing multidimensional arrays */
#include <stdio.h>

void printArray( const int a[][ 3 ] ); /* function prototype */

/* function main begins program execution */
int main( void )
{
   /* initialize array1, array2, array3 */
   int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
   int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
   int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };

   printf( "Values in array1 by row are:\n" );
   printArray( array1 );

   printf( "Values in array2 by row are:\n" );
   printArray( array2 );

   printf( "Values in array3 by row are:\n" );
   printArray( array3 );

   return 0; /* indicates successful termination */

} /* end main */
```

array1 is initialized with both rows full

array2 and **array3** are initialized only partially

- fig06_21.c (1 of 2)

301

```c
/* function to output array with two rows and three columns */
void printArray( const int a[][ 3 ] )
{
   int i; /* row counter */
   int j; /* column counter */

   /* loop through rows */
   for ( i = 0; i <= 1; i++ ) {

      /* output column values */
      for ( j = 0; j <= 2; j++ ) {
         printf( "%d ", a[ i ][ j ] );
      } /* end inner for */

      printf( "\n" ); /* start new line of output */
   } /* end outer for */

} /* end function printArray */
```

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

- fig06_21.c (2 of 2)

302

```c
/* Fig. 6.22: fig06_22.c
   Double-subscripted array example */
#include <stdio.h>
#define STUDENTS 3
#define EXAMS 4

/* function prototypes */
int minimum( const int grades[][ EXAMS ], int pupils, int tests );
int maximum( const int grades[][ EXAMS ], int pupils, int tests );
double average( const int setOfGrades[], int tests );
void printArray( const int grades[][ EXAMS ], int pupils, int tests );

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

   /* initialize student grades for three students (rows) */
   const int studentGrades[ STUDENTS ][ EXAMS ] =
      { { 77, 68, 86, 73 },
        { 96, 87, 89, 78 },
        { 70, 90, 86, 81 } };

   /* output array studentGrades */
   printf( "The array is:\n" );
   printArray( studentGrades, STUDENTS, EXAMS );

```

Each row in the array corresponds to a single student's set of grades

- fig06_22.c (1 of 6)

303

```c
   /* determine smallest and largest grade values */
   printf( "\n\nLowest grade: %d\nHighest grade: %d\n",
      minimum( studentGrades, STUDENTS, EXAMS ),
      maximum( studentGrades, STUDENTS, EXAMS ) );

   /* calculate average grade for each student */
   for ( student = 0; student < STUDENTS; student++ ) {
      printf( "The average grade for student %d is %.2f\n",
         student, average( studentGrades[ student ], EXAMS ) );
   } /* end for */

   return 0; /* indicates successful termination */

} /* end main */

```

average function is passed a row of the array

- fig06_22.c (2 of 6)

304

```c
/* Find the minimum grade */
int minimum( const int grades[][ EXAMS ], int pupils, int tests )
{
   int i; /* student counter */
   int j; /* exam counter */
   int lowGrade = 100; /* initialize to highest possible grade */

   /* loop through rows of grades */
   for ( i = 0; i < pupils; i++ ) {

      /* loop through columns of grades */
      for ( j = 0; j < tests; j++ ) {

         if ( grades[ i ][ j ] < lowGrade ) {
            lowGrade = grades[ i ][ j ];
         } /* end if */

      } /* end inner for */

   } /* end outer for */

   return lowGrade; /* return minimum grade */

} /* end function minimum */

```

- fig06_22.c (3 of 6)

305

```c
/* Find the maximum grade */
int maximum( const int grades[][ EXAMS ], int pupils, int tests )
{
   int i; /* student counter */
   int j; /* exam counter */
   int highGrade = 0; /* initialize to lowest possible grade */

   /* loop through rows of grades */
   for ( i = 0; i < pupils; i++ ) {

      /* loop through columns of grades */
      for ( j = 0; j < tests; j++ ) {

         if ( grades[ i ][ j ] > highGrade ) {
            highGrade = grades[ i ][ j ];
         } /* end if */

      } /* end inner for */

   } /* end outer for */

   return highGrade; /* return maximum grade */

} /* end function maximum */

```

- fig06_22.c (4 of 6)

306

```c
/* Determine the average grade for a particular student */
double average( const int setOfGrades[], int tests )
{
   int i; /* exam counter */
   int total = 0; /* sum of test grades */

   /* total all grades for one student */
   for ( i = 0; i < tests; i++ ) {
      total += setOfGrades[ i ];
   } /* end for */

   return ( double ) total / tests; /* average */

} /* end function average */

/* Print the array */
void printArray( const int grades[][ EXAMS ], int pupils, int tests )
{
   int i; /* student counter */
   int j; /* exam counter */

   /* output column heads */
   printf( "                 [0]  [1]  [2]  [3]" );

```

- fig06_22.c (5 of 6)

307

```c
   /* output grades in tabular format */
   for ( i = 0; i < pupils; i++ ) {

      /* output label for row */
      printf( "\nstudentGrades[%d] ", i );

      /* output grades for one student */
      for ( j = 0; j < tests; j++ ) {
         printf( "%-5d", grades[ i ][ j ] );
      } /* end inner for */

   } /* end outer for */

} /* end function printArray */
```

```
The array is:
                 [0]  [1]  [2]  [3]
studentGrades[0] 77   68   86   73
studentGrades[1] 96   87   89   78
studentGrades[2] 70   90   86   81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75
```

- fig06_22.c (6 of 6)

308

OBJECTIVES

- To define an array, initialize an array and refer to individual elements of an array.
- To define symbolic constants.
- To pass arrays to functions.
- To use arrays to store, sort and search lists and tables of values.
- To define and manipulate multiple-subscripted arrays.
- Review

309

Review

- Define an array
  - `arrayType arrayName[ numberOfElements ];`
- Symbolic constants
  - `#define SIZE 12 // no semicolon here!!!`
- Array size
  - The array subscript should never go below 0 and should always be less than the total number of elements in the array (size -1).
- Passing arrays to functions: passing by reference v.s. passing by values.
- Sorting arrays: bubble sort (or sinking sort).
- Searching arrays: linear search and binary search.
- Multiple-subscripted arrays

310

The End

- Thank you very much!

311

Lecture 7

C Pointers

Acknowledgment

The notes are adapted from those provided by Deitel & Associates, Inc. and Pearson Education Inc.

312

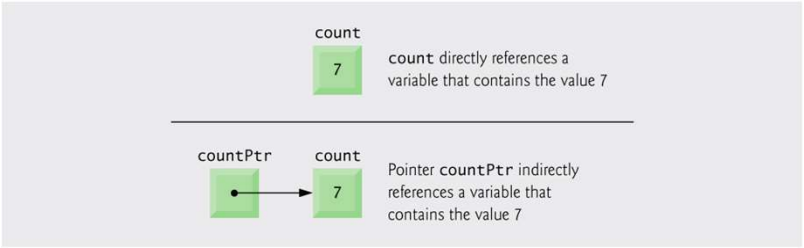
OBJECTIVES

- Pointer variable definitions and initialization
- Pointer operators
- Passing arguments to functions by reference
- Using const qualifier with pointers
- Bubble sort using call-by-reference
- Sizeof operator
- Pointer expressions and pointer arithmetic
- Relationships between pointers and arrays
- Array of pointers
- Case study: Card shuffling and dealing simulation
- To use pointers to functions

313

Pointer Variable Definitions and Initialization

- Contain memory addresses as their values
- Normal variables contain a specific value (direct reference)
- A pointer contains the address of a variable that has a specific value (indirect reference)



314

Pointer Variable Definitions and Initialization

- Pointer definitions
  - * used with pointer variables

    ```
    int *myPtr;
    ```
  - Defines a pointer to an `int` (pointer of type `int *`)
  - Multiple pointers require using a `*` before each variable definition

    ```
    int *myPtr1, *myPtr2;
    ```
  - Can define pointers to any data type
  - Initialize pointers to `0`, `NULL`, or an address
    - `0` or `NULL` – points to nothing
    - 0 is the only integer value that can be assigned directly to a pointer variable.
    - Initializing a pointer to 0 is equivalent to initializing a pointer to NULL, but NULL is preferred
    - NULL is a symbolic constant defined in the <stddef.h> header and several other headers, e.g. <stdio.h>

315

OBJECTIVES

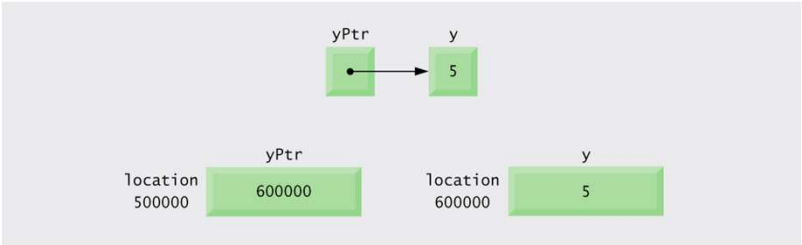
- Pointer variable definitions and initialization
- Pointer operators
- Passing arguments to functions by reference
- Using const qualifier with pointers
- Bubble sort using call-by-reference
- Sizeof operator
- Pointer expressions and pointer arithmetic
- Relationships between pointers and arrays
- Array of pointers
- Case study: Card shuffling and dealing simulation
- To use pointers to functions

316

Pointer Operators

- & (address operator)
  - Returns address of operand

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



317

Pointer Operators

- * (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand points to
  - `*yptr` returns `y` (because `yptr` points to `y`)
  - * can be used for assignment
    - Returns alias to an object

```
*yptr = 7;  /* changes y to 7 */
```

  - Dereferenced pointer (operand of *) must be an lvalue (no constants)
- * and & are inverses
  - They cancel each other out

318

```c
/* Fig. 7.4: fig07_04.c
   Using the & and * operators */
#include <stdio.h>

int main( void )
{
   int a;        /* a is an integer */
   int *aPtr;    /* aPtr is a pointer to an integer */

   a = 7;
   aPtr = &a;    /* aPtr set to address of a */

   printf( "The address of a is %p"
           "\nThe value of aPtr is %p", &a, aPtr );

   printf( "\n\nThe value of a is %d"
           "\nThe value of *aPtr is %d", a, *aPtr );

   printf( "\n\nShowing that * and & are complements of "
           "each other\n&*aPtr = %p"
           "\n*&aPtr = %p\n", &*aPtr, *&aPtr );

   return 0; /* indicates successful termination */

} /* end main */
```

- fig07_04.c (1 of 2)

If `aPtr` points to `a`, then `&a` and `aPtr` have the same value.

`a` and `*aPtr` have the same value

`&*aPtr` and `*&aPtr` have the same value

319

```
The address of a is 0012FF7C
The value of aPtr is 0012FF7C

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other.
&*aPtr = 0012FF7C
*&aPtr = 0012FF7C
```

- fig07_04.c (2 of 2)

320

Pointer Operator Precedence

Operators	Associativity	Type
( ) [ ]	left to right	highest
+ - ++ -- ! * & (*type*)	right to left	unary
* /	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	Equality
&&	left to right	logical and
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

321

OBJECTIVES

- Pointer variable definitions and initialization
- Pointer operators
- Passing arguments to functions by reference
- Using const qualifier with pointers
- Bubble sort using call-by-reference
- Sizeof operator
- Pointer expressions and pointer arithmetic
- Relationships between pointers and arrays
- Array of pointers
- Case study: Card shuffling and dealing simulation
- To use pointers to functions

322

Calling Functions by Reference

- 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

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

  - `*number` used as nickname for the variable passed

323

```c
/* Fig. 7.6: fig07_06.c
   Cube a variable using call-by-value */
#include <stdio.h>

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

int main( void )
{
   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 */

} /* end main */

/* 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 */
```

```
The original value of number is 5
The new value of number is 125
```

- fig07_06.c

324

```c
/* Fig. 7.7: fig07_07.c
   Cube a variable using call-by-reference with a pointer argument */

#include <stdio.h>

void cubeByReference( int *nPtr ); /* prototype */

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

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

   /* pass address of number to cubeByReference */
   cubeByReference( &number );

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

   return 0; /* indicates successful termination */

} /* end main */

/* calculate cube of *nPtr; modifies variable number in main */
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;  /* cube *nPtr */
} /* end function cubeByReference */
```

Function prototype takes a pointer argument

Function **cubeByReference** is passed an address, which can be the value of a pointer variable

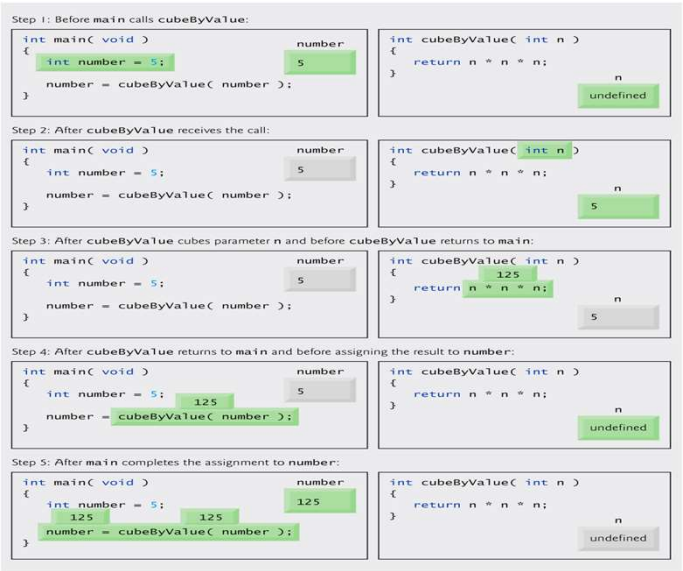
In this program, ***nPtr** is **number**, so this statement modifies the value of **number** itself.

```
The original value of number is 5
The new value of number is 125
```

- fig07_07.c

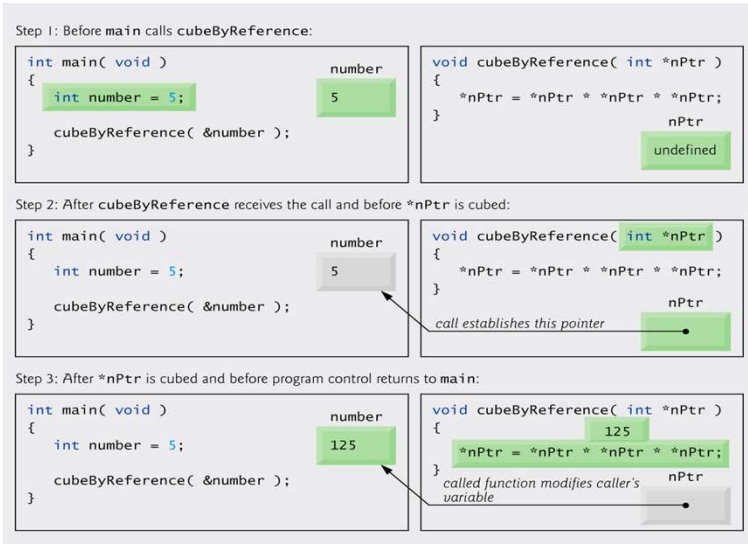
325

Analysis of A Typical Call-by-Value



326

Analysis of A Typical Call-by-Reference



327

OBJECTIVES

- Pointer variable definitions and initialization
- Pointer operators
- Passing arguments to functions by reference
- Using const qualifier with pointers
- Bubble sort using call-by-reference
- Sizeof operator
- Pointer expressions and pointer arithmetic
- Relationships between pointers and arrays
- Array of pointers
- Case study: Card shuffling and dealing simulation
- To use pointers to functions

328

Using the `const` Qualifier with Pointers

- `const` qualifier
  - Variable cannot be changed
  - Use `const` if function does not need to change a variable
  - Attempting to change a `const` variable produces an error
- `const` pointers
  - Point to a constant memory location
  - Must be initialized when defined
  - `int *const myPtr = &x;`
    - Type `int *const` – constant pointer to an `int`
  - `const int *myPtr = &x;`
    - Regular pointer to a `const int`
  - `const int *const Ptr = &x;`
    - `const` pointer to a `const int`
    - `x` can be changed, but not `*Ptr`

329

An example of using a non-constant pointer to non-

```
/* Fig. 7.10: fig07_10.c
   Converting lowercase letters to uppercase letters
   using a non-constant pointer to non-constant data */

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

void convertToUppercase( char *sPtr ); /* prototype */

int main( void )
{
   char string[] = "characters and $32.98"; /* initialize char array */

   printf( "The string before conversion is: %s", string );
   convertToUppercase( string );
   printf( "\nThe string after conversion is: %s\n", string );

   return 0; /* indicates successful termination */

} /* end main */
```

Both `sPtr` and `*sPtr` are modifiable

- fig07_10.c (1 of 2)

330

```c
/* convert string to uppercase letters */
void convertToUppercase( char *sPtr )
{
   while ( *sPtr != '\0' ) { /* current character is not '\0' */

      if ( islower( *sPtr ) ) {    /* if character is lowercase, */
         *sPtr = toupper( *sPtr ); /* convert to uppercase */
      } /* end if */

      ++sPtr;  /* move sPtr to the next character */
   } /* end while */

} /* end function convertToUppercase */
```

Both **sPtr** and ***sPtr** are modified by the **convertToUppercase** function

```
The string before conversion is: characters and $32.98
The string after conversion is: CHARACTERS AND $32.98
```

- fig07_10.c (2 of 2)

331

An example of using a non-constant pointer to constant data

```c
/* Fig. 7.11: fig07_11.c
   Printing a string one character at a time using
   a non-constant pointer to constant data */

#include <stdio.h>

void printCharacters( const char *sPtr );

int main( void )
{
   /* initialize char array */
   char string[] = "print characters of a string";

   printf( "The string is:\n" );
   printCharacters( string );
   printf( "\n" );

   return 0; /* indicates successful termination */

} /* end main */
```

Pointer variable **sPtr** is modifiable, but the data to which it points, ***sPtr**, is not

- fig07_11.c (1 of 2)

332

```c
/* sPtr cannot modify the character to which it points,
   i.e., sPtr is a "read-only" pointer */
void printCharacters( const char *sPtr )
{
   /* loop through entire string */
   for ( ; *sPtr != '\0'; sPtr++ ) { /* no initialization */
      printf( "%c", *sPtr );
   } /* end for */

} /* end function printCharacters */
```

sPtr is modified by function **printCharacters**

```
The string is:
print characters of a string
```

- fig07_11.c (2 of 2)

333

```c
/* Fig. 7.12: fig07_12.c
   Attempting to modify data through a
   non-constant pointer to constant data. */
#include <stdio.h>
void f( const int *xPtr ); /* prototype */


int main( void )
{
   int y;       /* define y */

   f( &y );     /* f attempts illegal modification */

   return 0;    /* indicates successful termination */

} /* end main */

/* xPtr cannot be used to modify the
   value of the variable to which it points */
void f( const int *xPtr )
{
   *xPtr = 100;  /* error: cannot modify a const object */
} /* end function f */
```

Pointer variable **xPtr** is modifiable, but the data to which it points, ***xPtr**, is not

***xPtr** has the **const** qualifier, so attempting to modify its value causes an error

```
Compiling...
FIG07_12.c
c:\books\2006\chtp5\examples\ch07\fig07_12.c(22) : error C2166: l-value
   specifies const object
Error executing cl.exe.

FIG07_12.exe - 1 error(s), 0 warning(s)
```

- fig07_12.c

334

```c
/* Fig. 7.13: fig07_13.c
   Attempting to modify a constant pointer to non-constant data */
#include <stdio.h>

int main( void )
{
   int x; /* define x */
   int y; /* define y */

   /* ptr is a constant pointer to an integer that can be modified
      through ptr, but ptr always points to the same memory location */
   int * const ptr = &x;

   *ptr = 7; /* allowed: *ptr is not const */
   ptr = &y; /* error: ptr is const; cannot assign new address */

   return 0; /* indicates successful termination */

} /* end main */
```

Pointer **ptr** is not modifiable, but the data to which it points, ***ptr**, can be changed

- fig07_13.c

```
Compiling...
FIG07_13.c
c:\books\2006\chtp5\Examples\ch07\FIG07_13.c(15) : error C2166: l-value
   specifies const object
Error executing cl.exe.

FIG07_13.exe - 1 error(s), 0 warning(s)
```

335

```c
/* Fig. 7.14: fig07_14.c
   Attempting to modify a constant pointer to constant data. */
#include <stdio.h>

int main( void )
{
   int x = 5; /* initialize x */
   int y;     /* define y */

   /* ptr is a constant pointer to a constant integer. ptr always
      points to the same location; the integer at that location
      cannot be modified */
   const int *const ptr = &x;

   printf( "%d\n", *ptr );

   *ptr = 7; /* error: *ptr is const; cannot assign new value */
   ptr = &y; /* error: ptr is const; cannot assign new address */

   return 0; /* indicates successful termination */

} /* end main */
```

Neither pointer **sPtr** nor the data to which it points, ***sPtr**, is modifiable

- fig07_14.c

```
Compiling...
FIG07_14.c
c:\books\2006\chtp5\Examples\ch07\FIG07_14.c(17) : error C2166: l-value
   specifies const object
c:\books\2006\chtp5\Examples\ch07\FIG07_14.c(18) : error C2166: l-value
   specifies const object
Error executing cl.exe.

FIG07_12.exe - 2 error(s), 0 warning(s)
```

336

OBJECTIVES

- Pointer variable definitions and initialization
- Pointer operators
- Passing arguments to functions by reference
- Using const qualifier with pointers
- Bubble sort using call-by-reference
- Sizeof operator
- Pointer expressions and pointer arithmetic
- Relationships between pointers and arrays
- Array of pointers
- Case study: Card shuffling and dealing simulation
- To use pointers to functions

337

Bubble Sort Using Call-by-reference

- Implement bubblesort using pointers
  - Swap two elements
  - `swap` function must receive address (using &) of array elements
    - Array elements have call-by-value default
  - Using pointers and the * operator, `swap` can switch array elements
- Psuedocode

  Initialize array

  print data in original order

  Call function bubblesort

  print sorted array

  Define bubblesort

338

```c
/* Fig. 7.15: fig07_15.c
   This program puts values into an array, sorts the values into
   ascending order, and prints the resulting array. */
#include <stdio.h>
#define SIZE 10

void bubbleSort( int * const array, const int size ); /* prototype */

int main( void )
{
   /* initialize array a */
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   int i; /* counter */

   printf( "Data items in original order\n" );

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

   bubbleSort( a, SIZE ); /* sort the array */

   printf( "\nData items in ascending order\n" );

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

- fig07_15.c (1 of 3)

339

```c

   printf( "\n" );

   return 0; /* indicates successful termination */

} /* end main */

/* sort an array of integers using bubble sort algorithm */
void bubbleSort( int * const array, const int size )
{
   void swap( int *element1Ptr, int *element2Ptr ); /* prototype */
   int pass; /* pass counter */
   int j;    /* comparison counter */

   /* loop to control passes */
   for ( pass = 0; pass < size - 1; pass++ ) {

      /* loop to control comparisons during each pass */
      for ( j = 0; j < size - 1; j++ ) {

         /* swap adjacent elements if they are out of order */
         if ( array[ j ] > array[ j + 1 ] ) {
            swap( &array[ j ], &array[ j + 1 ] );
         } /* end if */

      } /* end inner for */

   } /* end outer for */

} /* end function bubbleSort */
```

- fig07_15.c (2 of 3)

340

```c

/* swap values at memory locations to which element1Ptr and
   element2Ptr point */
void swap( int *element1Ptr, int *element2Ptr )
{
   int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
} /* end function swap */
```

Function **swap** changes the values of the **int**s that the two pointers point to

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

- fig07_15.c (3 of 3)

341

OBJECTIVES

- Pointer variable definitions and initialization
- Pointer operators
- Passing arguments to functions by reference
- Using const qualifier with pointers
- Bubble sort using call-by-reference
- Sizeof operator
- Pointer expressions and pointer arithmetic
- Relationships between pointers and arrays
- Array of pointers
- Case study: Card shuffling and dealing simulation
- To use pointers to functions

342

sizeof

- `sizeof`
  - Returns size of operand in bytes
  - For arrays: size of 1 element * number of elements
  - if `sizeof( int )` equals 4 bytes, then

```
int myArray[ 10 ];
printf( "%d", sizeof( myArray ) );
```

    - will print 40
- `sizeof` can be used with
  - Variable names
  - Type name
  - Constant values

343

```c
/* Fig. 7.16: fig07_16.c
   Sizeof operator when used on an array name
   returns the number of bytes in the array. */
#include <stdio.h>

size_t getSize( float *ptr ); /* prototype */

int main( void )
{
   float array[ 20 ]; /* create array */

   printf( "The number of bytes in the array is %d"
           "\nThe number of bytes returned by getSize is %d\n",
           sizeof( array ), getSize( array ) );

   return 0; /* indicates successful termination */

} /* end main */

/* return size of ptr */
size_t getSize( float *ptr )
{
   return sizeof( ptr );

} /* end function getSize */
```

`float`s take up 4 bytes in memory, so 20 `float`s take up 80 bytes

```
The number of bytes in the array is 80
The number of bytes returned by getSize is 4
```

- fig07_16.c

344

```c
/* Fig. 7.17: fig07_17.c
   Demonstrating the sizeof operator */
#include <stdio.h>

int main( void )
{
   char c;
   short s;
   int i;
   long l;
   float f;
   double d;
   long double ld;
   int array[ 20 ];  /* create array of 20 int elements */
   int *ptr = array; /* create pointer to array */

```

- fig07_17.c (1 of 2)

345

```c
   printf( "     sizeof c = %d\tsizeof(char)  = %d"
           "\n     sizeof s = %d\tsizeof(short) = %d"
           "\n     sizeof i = %d\tsizeof(int) = %d"
           "\n     sizeof l = %d\tsizeof(long) = %d"
           "\n     sizeof f = %d\tsizeof(float) = %d"
           "\n     sizeof d = %d\tsizeof(double) = %d"
           "\n    sizeof ld = %d\tsizeof(long double) = %d"
           "\n sizeof array = %d"
           "\n   sizeof ptr = %d\n",
          sizeof c, sizeof( char ), sizeof s, sizeof( short ), sizeof i,
          sizeof( int ), sizeof l, sizeof( long ), sizeof f,
          sizeof( float ), sizeof d, sizeof( double ), sizeof ld,
          sizeof( long double ), sizeof array, sizeof ptr );

   return 0; /* indicates successful termination */

} /* end main */
```

```
     sizeof c = 1       sizeof(char)  = 1
     sizeof s = 2       sizeof(short) = 2
     sizeof i = 4       sizeof(int) = 4
     sizeof l = 4       sizeof(long) = 4
     sizeof f = 4       sizeof(float) = 4
     sizeof d = 8       sizeof(double) = 8
    sizeof ld = 8       sizeof(long double) = 8
 sizeof array = 80
   sizeof ptr = 4
```

- fig07_17.c (2 of 2)

346

OBJECTIVES

- Pointer variable definitions and initialization
- Pointer operators
- Passing arguments to functions by reference
- Using const qualifier with pointers
- Bubble sort using call-by-reference
- Sizeof operator
- Pointer expressions and pointer arithmetic
- Relationships between pointers and arrays
- Array of pointers
- Case study: Card shuffling and dealing simulation
- To use pointers to functions

347

Pointer Expressions and Pointer Arithmetic

- Arithmetic operations can be performed on pointers
  - Increment/decrement pointer (++ or --)
  - Add an integer to a pointer(+ or += , - or -=)
  - Pointers may be subtracted from each other
  - Operations meaningless unless performed on an array

348

Pointer Expressions and Pointer Arithmetic

- 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`

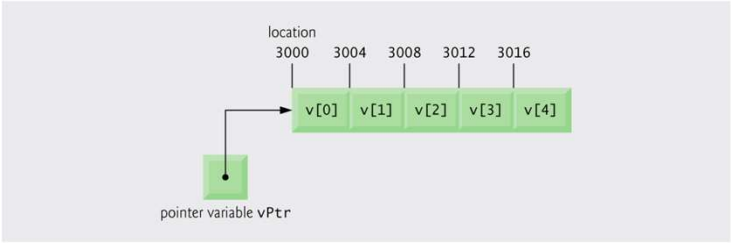
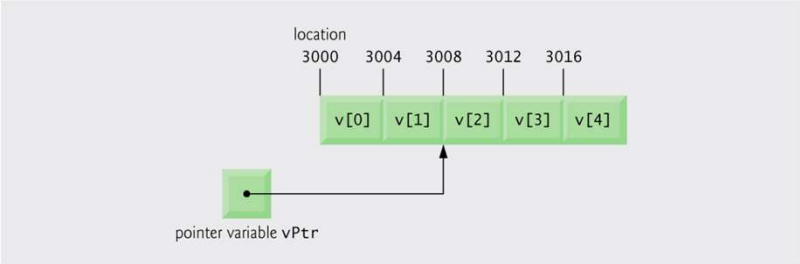


Fig. 7.18 | Array **v** and a pointer variable **vPtr** that points to **v**.

349

The pointer **vPtr** after pointer arithmetic



350

Pointer Expressions and Pointer Arithmetic

- Subtracting pointers
  - Returns number of elements from one to the other. If
    `vPtr2 = v[ 2 ];`
    `vPtr = v[ 0 ];`
  - `vPtr2 - vPtr` would produce 2
- Pointer comparison (<, == , >)
  - See which pointer points to the higher numbered array element
  - Also, see if a pointer points to `0`

351

Pointer Expressions and Pointer Arithmetic

- 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

352

OBJECTIVES

- Pointer variable definitions and initialization
- Pointer operators
- Passing arguments to functions by reference
- Using const qualifier with pointers
- Bubble sort using call-by-reference
- Sizeof operator
- Pointer expressions and pointer arithmetic
- Relationships between pointers and arrays
- Array of pointers
- Case study: Card shuffling and dealing simulation
- To use pointers to functions

353

The Relationship Between Pointers and Arrays

- 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`

354

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 )`

355

```c
/* Fig. 7.20: fig07_20.cpp
   Using subscripting and pointer notations with arrays */

#include <stdio.h>

int main( void )
{
   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:\nArray 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\n"
           "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 */

```

Array subscript notation

Pointer/offset notation

- fig07_20.c (1 of 3)

356

```c
   /* 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 */
```

Pointer subscript notation

Pointer offset notation

```
Array b printed with:
Array subscript notation
b[ 0 ] = 10
b[ 1 ] = 20
b[ 2 ] = 30
b[ 3 ] = 40
```

(continued on next slide…)

- fig07_20.c (2 of 3)

357

(continued from previous slide…)

```
Pointer/offset notation where
the pointer is the array name
*( b + 0 ) = 10
*( b + 1 ) = 20
*( b + 2 ) = 30
*( b + 3 ) = 40

Pointer subscript notation
bPtr[ 0 ] = 10
bPtr[ 1 ] = 20
bPtr[ 2 ] = 30
bPtr[ 3 ] = 40

Pointer/offset notation
*( bPtr + 0 ) = 10
*( bPtr + 1 ) = 20
*( bPtr + 2 ) = 30
*( bPtr + 3 ) = 40
```

- fig07_20.c (3 of 3)

358

```c
/* Fig. 7.21: fig07_21.c
   Copying a string using array notation and pointer notation. */
#include <stdio.h>

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

int main( void )
{
   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 */

```

- fig07_21.c (1 of 2)

359

```c
/* copy s2 to s1 using array notation */
void copy1( char * const s1, const char * const 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 */
```

Condition of **for** loop actually performs an action

```
string1 = Hello
string3 = Good Bye
```

- fig07_21.c (2 of 2)

360

OBJECTIVES

- Pointer variable definitions and initialization
- Pointer operators
- Passing arguments to functions by reference
- Using const qualifier with pointers
- Bubble sort using call-by-reference
- Sizeof operator
- Pointer expressions and pointer arithmetic
- Relationships between pointers and arrays
- Array of pointers
- Case study: Card shuffling and dealing simulation
- To use pointers to functions

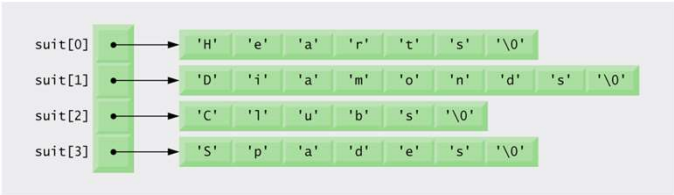
361

Arrays of Pointers

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

```
char *suit[ 4 ] = { "Hearts", "Diamonds",
    "Clubs", "Spades" };
```

  - 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

362

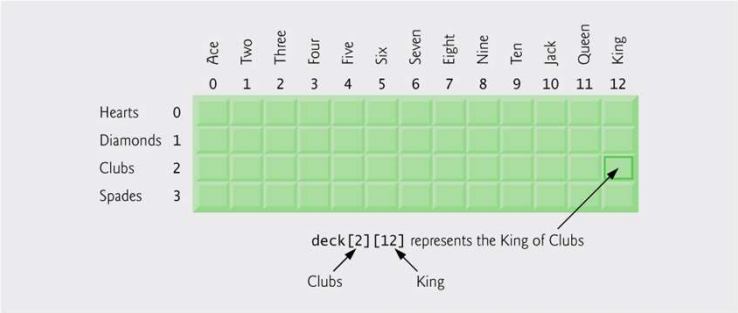
Case study: Roman numeral equivalents

```
#include <stdio.h>

void main( void ) {
        int decimal_number = 101, a = 0, b = 0;
        const char *x[11] = {"", "x", "xx", "xxx", "xl", "l", "lx", "lxx", "lxxx", "xc",
"c"};
        const char *y[10] = {"", "i", "ii", "iii", "iv", "v", "vi", "vii", "viii", "ix"};

        while ((decimal_number > 100) || (decimal_number < 0)) {
                printf("Enter the decimal numbers in the range 1 to 100:\n");
                scanf("%d", &decimal_number);
        }

        a = decimal_number/10;
        b = decimal_number%10;

        printf("The equivalent roman is %s%s\n", x[a], y[b]);
}
```

363

Case Study: A Card Shuffling and Dealing Simulation

- 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

364

Case Study: A Card Shuffling and Dealing Simulation

- 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

365

Case Study: Card Shuffling and Dealing Simulation

- 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

366

Case Study: A Card Shuffling and Dealing Simulation

- Third refinement
  - Convert *shuffle the deck* to
    Choose slot of deck randomly
    While chosen slot of deck has been previously chosen
    Choose slot of deck randomly
    Place card number in chosen slot of deck
  - Convert *deal 52 cards* to
    For each slot of the deck array
    If slot contains card number
    Print the face and suit of the card

367

```c
/* 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( void )
{
   /* initialize suit array */
   const char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };

   /* initialize face array */
   const char *face[ 13 ] =
      { "Ace", "Deuce", "Three", "Four",
        "Five", "Six", "Seven", "Eight",
        "Nine", "Ten", "Jack", "Queen", "King" };

```

suit and **face** arrays are arrays of pointers

- fig07_24.c (1 of 4)

368

```c
   /* 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 */

```

- fig07_24.c (2 of 4)

do...while loop selects a random spot for each card

369

```c
      /* 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++ ) {
```

- fig07_24.c (3 of 4)

370

```c

            /* if slot contains current card, display card */
            if ( wDeck[ row ][ column ] == card ) {
               printf( "%5s of %-8s%c", wFace[ column ], wSuit[ row ],
                  card % 2 == 0 ? '\n' : '\t' );
            } /* end if */

         } /* end for */

      } /* end for */

   } /* end for */

} /* end function deal */
```

- fig07_24.c (4 of 4)

371

```
Nine of Hearts          Five of Clubs
Queen of Spades         Three of Spades
Queen of Hearts           Ace of Clubs
 King of Hearts           Six of Spades
 Jack of Diamonds        Five of Spades
Seven of Hearts          King of Clubs
Three of Clubs          Eight of Hearts
Three of Diamonds        Four of Diamonds
Queen of Diamonds        Five of Diamonds
  Six of Diamonds        Five of Hearts
  Ace of Spades           Six of Hearts
 Nine of Diamonds        Queen of Clubs
Eight of Spades          Nine of Clubs
Deuce of Clubs            Six of Clubs
Deuce of Spades          Jack of Clubs
 Four of Clubs           Eight of Clubs
 Four of Spades          Seven of Spades
Seven of Diamonds        Seven of Clubs
 King of Spades            Ten of Diamonds
 Jack of Hearts            Ace of Hearts
 Jack of Spades            Ten of Clubs
Eight of Diamonds        Deuce of Diamonds
  Ace of Diamonds         Nine of Spades
 Four of Hearts          Deuce of Hearts
 King of Diamonds          Ten of Spades
Three of Hearts            Ten of Hearts
```

372

OBJECTIVES

- Pointer variable definitions and initialization
- Pointer operators
- Passing arguments to functions by reference
- Using const qualifier with pointers
- Bubble sort using call-by-reference
- Sizeof operator
- Pointer expressions and pointer arithmetic
- Relationships between pointers and arrays
- Array of pointers
- Case study: Card shuffling and dealing simulation
- To use pointers to functions

373

Pointers to Functions

- Pointer to function
  - Contains address of function
  - Similar to how array name is address of first element
  - Function name is starting address of code that defines function
- Function pointers can be
  - Passed to functions
  - Stored in arrays
  - Assigned to other function pointers

374

Pointers to Functions

- Example: bubblesort
  - Function `bubble` takes a function pointer
    - `bubble` calls this helper function
    - this determines ascending or descending sorting
  - The argument in `bubble` for the function pointer:

    `int ( *compare )( int a, int b )`

    tells `bubble` to expect a pointer to a function that takes two `ints` and returns an `int`
  - If the parentheses were left out:

    `int *compare( int a, int b )`
    - Defines a function that receives two integers and returns a pointer to a `int`

375

```c
/* Fig. 7.26: fig07_26.c
   Multipurpose sorting program using function pointers */
#include <stdio.h>
#define SIZE 10

/* prototypes */
void bubble( int work[], const int size, int (*compare)( int a, int b ) );
int ascending( int a, int b );
int descending( int a, int b );

int main( void )
{
   int order;   /* 1 for ascending order or 2 for descending order */
   int counter; /* counter */

   /* initialize array a */
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   printf( "Enter 1 to sort in ascending order,\n"
           "Enter 2 to sort in descending order: " );
   scanf( "%d",  &order );

   printf( "\nData items in original order\n" );

   /* output original array */
   for ( counter = 0; counter < SIZE; counter++ ) {
      printf( "%5d", a[ counter ] );
   } /* end for */

```

`bubble` function takes a function pointer as an argument

- fig07_26.c (1 of 4)

376

```c
   /* sort array in ascending order; pass function ascending as an
     argument to specify ascending sorting order */
   if ( order == 1 ) {
      bubble( a, SIZE, ascending );
      printf( "\nData items in ascending order\n" );
   } /* end if */
   else { /* pass function descending */
      bubble( a, SIZE, descending );
      printf( "\nData items in descending order\n" );
   } /* end else */

   /* output sorted array */
   for ( counter = 0; counter < SIZE; counter++ ) {
      printf( "%5d", a[ counter ] );
   } /* end for */

   printf( "\n" );

   return 0; /* indicates successful termination */

} /* end main */

```

depending on the user's choice, the **bubble** function uses either the **ascending** or **descending** function to sort the array

- fig07_26.c (2 of 4)

377

```c
/* multipurpose bubble sort; parameter compare is a pointer to
   the comparison function that determines sorting order */
void bubble( int work[], const int size, int (*compare)( int a, int b ) )
{
   int pass;  /* pass counter */
   int count; /* comparison counter */

   void swap( int *element1Ptr, int *element2ptr ); /* prototype */

   /* loop to control passes */
   for ( pass = 1; pass < size; pass++ ) {

      /* loop to control number of comparisons per pass */
      for ( count = 0; count < size - 1; count++ ) {

         /* if adjacent elements are out of order, swap them */
         if ( (*compare)( work[ count ], work[ count + 1 ] ) ) {
            swap( &work[ count ], &work[ count + 1 ] );
         } /* end if */

      } /* end for */

   } /* end for */

} /* end function bubble */

```

Note that what the program considers "out of order" is dependent on the function pointer that was passed to the **bubble** function

- fig07_26.c (3 of 4)

378

```c
/* swap values at memory locations to which element1Ptr and
   element2Ptr point */
void swap( int *element1Ptr, int *element2Ptr )
{
   int hold; /* temporary holding variable */

   hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
} /* end function swap */

/* determine whether elements are out of order for an ascending
   order sort */
int ascending( int a, int b )
{
   return b < a; /* swap if b is less than a */

} /* end function ascending */

/* determine whether elements are out of order for a descending
   order sort */
int descending( int a, int b )
{
   return b > a; /* swap if b is greater than a */

} /* end function descending */
```

- fig07_26.c (4 of 4)

Passing the **bubble** function **ascending** will point the program here

Passing the **bubble** function **descending** will point the program here

379

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in ascending order
    2    4    6    8   10   12   37   45   68   89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in descending order
   89   68   45   37   12   10    8    6    4    2
```

380

Review

- ★ Pointers: data type * pointer variable name (%p, hexadecimal integer)
- ★ Call-by-value (not change) and call-by-reference (change)
- ★ Const qualifier make variables not be modified (four combinations)
- ★ Sizeof can determine the size in bytes
- ★ Arrays of pointers and case study
- Pointer to function: the name of the function is the address

381

The End

- Thank you very much!

382

Chapter 8

Characters and Strings

Acknowledgment

The notes are adapted from those provided by Deitel & Associates, Inc. and Pearson Education Inc.

383

OBJECTIVES

- Fundamentals of strings and characters
- Character-handling library
- String-conversion functions
- Standard input/output library functions
- String-manipulation functions of the string-handling library
  - Comparison functions of the string-handling library
  - Search functions of the string-handling library
  - Other functions of the string-handling library
- Review

384

Introduction

- Introduce some standard library functions
  - Easy string and character processing
  - Programs can process characters, strings, lines of text, and blocks of memory
- These techniques used to make
  - Word processors
  - Page layout software
  - Typesetting programs

385

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

386

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'`

387

OBJECTIVES

- Fundamentals of strings and characters
- Character-handling library
- String-conversion functions
- Standard input/output library functions
- String-manipulation functions of the string-handling library
  - Comparison functions of the string-handling library
  - Search functions of the string-handling library
  - Other functions of the string-handling library
- Review

388

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>`

389

Character-handling library functions

Prototype	Description
`int isdigit( int c );`	Returns `true` if `c` is a digit and `false` otherwise.
`int isalpha( int c );`	Returns `true` if `c` is a letter and `false` otherwise.
`int isalnum( int c );`	Returns `true` if `c` is a digit or a letter and `false` otherwise.
`int isxdigit( int c );`	Returns `true` if `c` is a hexadecimal digit character and `false` otherwise.
`int islower( int c );`	Returns `true` if `c` is a lowercase letter and `false` otherwise.
`int isupper( int c );`	Returns `true` if `c` is an uppercase letter; `false` otherwise.
`int tolower( int c );`	If `c` is an uppercase letter, `tolower` returns `c` as a lowercase letter. Otherwise, `tolower` returns the argument unchanged.
`int toupper( int c );`	If `c` is a lowercase letter, `toupper` returns `c` as an uppercase letter. Otherwise, `toupper` returns the argument unchanged.
`int isspace( int c );`	Returns `true` if `c` is a white-space character—newline (`'\n'`), space (`' '`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`), or vertical tab (`'\v'`)—and `false` otherwise
`int iscntrl( int c );`	Returns `true` if `c` is a control character and `false` otherwise.
`int ispunct( int c );`	Returns `true` if `c` is a printing character other than a space, a digit, or a letter and `false` otherwise.
`int isprint( int c );`	Returns `true` value if `c` is a printing character including space (`' '`) and `false` otherwise.
`int isgraph( int c );`	Returns `true` if `c` is a printing character other than space (`' '`) and `false` otherwise.

390

```c
/* Fig. 8.2: fig08_02.c
   Using functions isdigit, isalpha, isalnum, and isxdigit */
#include <stdio.h>
#include <ctype.h>

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

   printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
       "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" );

```

isdigit tests if a character is a decimal digit

isalpha tests if a character is a letter

- fig08_02.c (1 of 3)

391

```c
   printf( "%s\n%s%s\n%s%s\n%s%s\n\n",
       "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( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s\n",
       "According to isxdigit:",
       isxdigit( 'F' ) ? "F is a " : "F is not a ",
       "hexadecimal digit",
       isxdigit( 'J' ) ? "J is a " : "J is not a ",
       "hexadecimal digit",
       isxdigit( '7' ) ? "7 is a " : "7 is not a ",
       "hexadecimal digit",
       isxdigit( '$' ) ? "$ is a " : "$ is not a ",
```

isdigit tests if a character is a decimal digit or a letter

isxdigit tests if a character is a hexadecimal digit

- fig08_02.c (2 of 3)

392

```c
      "hexadecimal digit",
      isxdigit( 'f' ) ? "f is a " : "f is not a ",
      "hexadecimal digit" );

   return 0; /* indicates successful termination */

} /* end main */
```

```
According to isdigit:
8 is a digit
# is not a digit

According to isalpha:
A is a letter
b is a letter
& is not a letter
4 is not a letter

According to isalnum:
A is a digit or a letter
8 is a digit or a letter
# is not a digit or a letter

According to isxdigit:
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is a hexadecimal digit
```

- fig08_02.c (3 of 3)

393

```c
/* Fig. 8.3: fig08_03.c
   Using functions islower, isupper, tolower, toupper */
#include <stdio.h>
#include <ctype.h>

int main( void )
{
   printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
           "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( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
           "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" );

```

- fig08_03.c (1 of 2)

islower tests if a character is a lowercase letter

isupper tests if a character is an uppercase letter

394

```c
   printf( "%s%c\n%s%c\n%s%c\n%s%c\n",
           "u converted to uppercase is ", toupper( 'u' ),
           "7 converted to uppercase is ", toupper( '7' ),
           "$ converted to uppercase is ", toupper( '$' ),
           "L converted to lowercase is ", tolower( 'L' ) );

   return 0; /* indicates successful termination */

} /* end main */
```

toupper and **tolower** convert letters to upper or lower case

```
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
```

- fig08_03.c (2 of 2)

395

```c
/* Fig. 8.4: fig08_04.c
  Using functions isspace, iscntrl, ispunct, isprint, isgraph */
#include <stdio.h>
#include <ctype.h>

int main( void )
{
   printf( "%s\n%s%s%s\n%s%s%s\n%s%s\n\n",
       "According to isspace:",
       "Newline", isspace( '\n' ) ? " is a " : " is not a ",
       "whitespace character", "Horizontal tab",
       isspace( '\t' ) ? " is a " : " is not a ",
       "whitespace character",
       isspace( '%' ) ? "% is a " : "% is not a ",
       "whitespace character" );

   printf( "%s\n%s%s%s\n%s%s\n\n", "According to iscntrl:",
       "Newline", iscntrl( '\n' ) ? " is a " : " is not a ",
       "control character", iscntrl( '$' ) ? "$ is a " :
       "$ is not a ", "control character" );
```

isspace tests if a character is a whitespace character

iscntrl tests if a character is a control character

- fig08_04.c (1 of 3)

396

```c
   printf( "%s\n%s%s\n%s%s\n%s%s\n\n",
       "According to ispunct:",
       ispunct( ';' ) ? "; is a " : "; is not a ",
       "punctuation character",
       ispunct( 'Y' ) ? "Y is a " : "Y is not a ",
       "punctuation character",
       ispunct( '#' ) ? "# is a " : "# is not a ",
       "punctuation character" );

   printf( "%s\n%s%s\n%s%s%s\n\n", "According to isprint:",
       isprint( '$' ) ? "$ is a " : "$ is not a ",
       "printing character",
       "Alert", isprint( '\a' ) ? " is a " : " is not a ",
       "printing character" );
```

ispunct tests if a character is a punctuation character

isprint tests if a character is a printing character

- fig08_04.c (2 of 3)

397

```c
   printf( "%s\n%s%s\n%s%s%s\n",  "According to isgraph:",
       isgraph( 'Q' ) ? "Q is a " : "Q is not a ",
       "printing character other than a space",
       "Space", isgraph( ' ' ) ? " is a " : " is not a ",
       "printing character other than a space" );

   return 0; /* indicates successful termination */

} /* end main */
```

isgraph tests if a character is a printing character that is not a space

```
According to isspace:
Newline is a whitespace character
Horizontal tab is a whitespace character
% is not a whitespace character

According to iscntrl:
Newline is a control character
$ is not a control character

According to ispunct:
; is a punctuation character
Y is not a punctuation character
# is a punctuation character

According to isprint:
$ is a printing character
Alert is not a printing character

According to isgraph:
Q is a printing character other than a space
Space is not a printing character other than a space
```

- fig08_04.c (3 of 3)

398

OBJECTIVES

- Fundamentals of strings and characters
- Character-handling library
- ★ String-conversion functions
- Standard input/output library functions
- String-manipulation functions of the string-handling library
  - Comparison functions of the string-handling library
  - Search functions of the string-handling library
  - Other functions of the string-handling library
- Review

399

String Conversion Functions

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

Function prototype	Function description
`double atof( const char *nPtr );`	Converts the string `nPtr` to `double`.
`int atoi( const char *nPtr );`	Converts the string `nPtr` to `int`.
`long atol( const char *nPtr );`	Converts the string `nPtr` to `long int`.
`double strtod( const char *nPtr, char **endPtr );`	Converts the string `nPtr` to `double`.
`long strtol( const char *nPtr, char **endPtr, int base );`	Converts the string `nPtr` to `long`.
`unsigned long strtoul( const char *nPtr, char **endPtr, int base );`	Converts the string `nPtr` to `unsigned long`.

400

```c
/* Fig. 8.6: fig08_06.c
   Using atof */
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   double d; /* variable to hold converted string */

   d = atof( "99.0" );

   printf( "%s%.3f\n%s%.3f\n",
           "The string \"99.0\" converted to double is ", d,
           "The converted value divided by 2 is ",
           d / 2.0 );

   return 0; /* indicates successful termination */

} /* end main */
```

atof converts a string to a **double**

```
The string "99.0" converted to double is 99.000
The converted value divided by 2 is 49.500
```

- fig08_06.c

Note:

If the converted value cannot be represented, e.g., if the first character of the string is a letter, the behavior of function *atof* is undefined.

401

```c
/* Fig. 8.7: fig08_07.c
   Using atoi */
#include <stdio.h>
#include <stdlib.h>

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

   i = atoi( "2593" );

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

   return 0; /* indicates successful termination */

} /* end main */
```

atoi converts a string to an **int**

```
The string "2593" converted to int is 2593
The converted value minus 593 is 2000
```

- fig08_07.c

402

```c
/* Fig. 8.8: fig08_08.c
   Using atol */
#include <stdio.h>
#include <stdlib.h>

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

   l = atol( "1000000" );

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

   return 0; /* indicates successful termination */

} /* end main */
```

`atol` converts a string to a `long`

```
The string "1000000" converted to long int is 1000000
The converted value divided by 2 is 500000
```

- fig08_08.c

403

```c
/* Fig. 8.9: fig08_09.c
   Using strtod */
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   /* initialize string pointer */
   const char *string = "51.2% are admitted"; /* initialize string */

   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 */
```

`strtod` converts a piece of a string to a `double`

```
The string "51.2% are admitted" is converted to the
double value 51.20 and the string "% are admitted"
```

- fig08_09.c

404

```c
/* Fig. 8.10: fig08_10.c
   Using strtol */
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   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( "%s\"%s\"\n%s%ld\n%s\"%s\"\n%s%ld\n",
           "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
```

- fig08_10.c

`strtol` converts a piece of a string to a `long`

Note:

1. Using NULL for the second argument causes the remainder of the string to be ignored.

2. The third argument, 0, indicates that the value to be converted can be in octal (base 8), decimal (base 10) or hexadecimal (base 16) format.

405

```c
/* Fig. 8.11: fig08_11.c
   Using strtoul */
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   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( "%s\"%s\"\n%s%lu\n%s\"%s\"\n%s%lu\n",
           "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 */

} /* end main */
```

```
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
```

- fig08_11.c

`strtoul` converts a piece of a string to an `unsigned long`

406

OBJECTIVES

- Fundamentals of strings and characters
- Character-handling library
- String-conversion functions
- Standard input/output library functions
- String-manipulation functions of the string-handling library
  - Comparison functions of the string-handling library
  - Search functions of the string-handling library
  - Other functions of the string-handling library
- Review

407

Standard Input/Output Library Functions

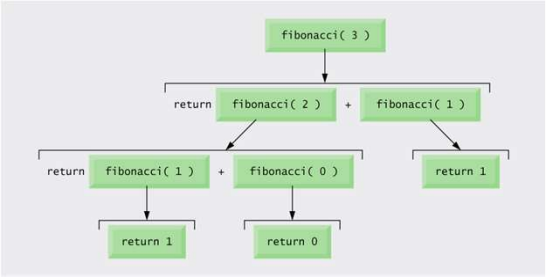
- Functions in <stdio.h>
- Used to manipulate character and string data

Function prototype	Function description
`int getchar( void );`	Inputs the next character from the standard input and returns it as an integer.
`char *gets( char *s );`	Inputs characters from the standard input into the array `s` until a newline or end-of-file character is encountered. A terminating null character is appended to the array. Returns the string inputted into `s`. Note that an error will occur if `s` is not large enough to hold the string.
`int putchar( int c );`	Prints the character stored in `c` and returns it as an integer.
`int puts( const char *s );`	Prints the string `s` followed by a newline character. Returns a non-zero integer if successful, or `EOF` if an error occurs.
`int sprintf( char *s, const char *format, ... );`	Equivalent to `printf`, except the output is stored in the array `s` instead of printed on the screen. Returns the number of characters written to `s`, or `EOF` if an error occurs.
`int sscanf( char *s, const char *format, ... );`	Equivalent to `scanf`, except the input is read from the array `s` rather than from the keyboard. Returns the number of items successfully read by the function, or `EOF` if an error occurs.

408

Recursive Functions

- The function calls itself in its body.
- Do itself again and again until reach the base case.
- Analyze actions by diagrams.



409

```c
/* Fig. 8.13: fig08_13.c
   Using gets and putchar */
#include <stdio.h>

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

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

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

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

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

   return 0; /* indicates successful termination */

} /* end main */
```

gets reads a line of text from the user

- fig08_13.c
  (1 of 2)

410

```c

/* recursively outputs characters in string in reverse order */
void reverse( const char * const sPtr )
{
   /* if end of the string */
   if ( sPtr[ 0 ] == '\0' ) { /* base case */
      return;
   } /* end if */
   else { /* if not end of the string */
      reverse( &sPtr[ 1 ] ); /* recursion step */

      putchar( sPtr[ 0 ] ); /* use putchar to display character */
   } /* end else */

} /* end function reverse */
```

putchar prints a single character on the screen

```
Enter a line of text:
Characters and Strings

The line printed backward is:
sgnirtS dna sretcarahC
```

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

The line printed backward is:
able was I ere I saw elba
```

- fig08_13.c

(2 of 2)

411

Detailed Analysis

- Step 1: reverse("abc")
  - Step 2: reverse(&sPtr[1]) → reverse("bc") (Do first)
  - putchar(sPtr[0]) → putchar('a') (Do next)
- Step 2: reverse("bc")
  - Step 3: reverse(&sPtr[1]) → reverse("c") (Do first)
  - putchar(sPtr[0]) → putchar('b') (Do next)
- Step 3: reverse("c")
  - Step 4: reverse(&sPtr[1]) → reverse("\0") (Do first)
  - putchar(sPtr[0]) → putchar('c') (Do next)
- Step 4: reverse("\0")
  - base case → return
- Step 4 → Step 3 → Step 2 → Step 1 "cba"

412

```c
/* Fig. 8.14: fig08_14.c
   Using getchar and puts */
#include <stdio.h>

int main( void )
{
   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'; /* terminate string */

```

puts prints a line of text on the screen

getchar reads a single character from the user

- fig08_14.c (1 of 2)

413

```c
   /* use puts to display sentence */
   puts( "\nThe line entered was:" );
   puts( sentence );

   return 0; /* indicates successful termination */

} /* end main */
```

```
Enter a line of text:
This is a test.

The line entered was:
This is a test.
```

- fig08_14.c **(2 of 2)**

414

```c
/* Fig. 8.15: fig08_15.c
   Using sprintf */
#include <stdio.h>

int main( void )
{
   char s[ 80 ]; /* create char array */
   int x;        /* x value to be input */
   double y;     /* y value to be input */

   printf( "Enter an integer and a double:\n" );
   scanf( "%d%lf", &x, &y );

   sprintf( s, "integer:%6d\ndouble:%8.2f", x, y );

   printf( "%s\n%s\n",
           "The formatted output stored in array s is:", s );

   return 0; /* indicates successful termination */

} /* end main */
```

```
Enter an integer and a double:
298 87.375
The formatted output stored in array s is:
integer:    298
double:   87.38
```

- fig08_15.c

sprintf prints a line of text into an array like **printf** prints text on the screen

415

```c
/* Fig. 8.16: fig08_16.c
   Using sscanf */
#include <stdio.h>

int main( void )
{
   char s[] = "31298 87.375"; /* initialize array s */
   int x;    /* x value to be input */
   double y; /* y value to be input */

   sscanf( s, "%d%lf", &x, &y );

   printf( "%s\n%s%6d\n%s%8.3f\n",
           "The values stored in character array s are:",
           "integer:", x, "double:", y );

   return 0; /* indicates successful termination */

} /* end main */
```

```
The values stored in character array s are:
integer: 31298
double:  87.375
```

- fig08_16.c

sscanf reads a line of text from an array like **scanf** reads text from the user

416

OBJECTIVES

- Fundamentals of strings and characters
- Character-handling library
- String-conversion functions
- Standard input/output library functions
- String-manipulation functions of the string-handling library
  - Comparison functions of the string-handling library
  - Search functions of the string-handling library
  - Other functions of the string-handling library
- Review

417

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	Function description
`char *strcpy( char *s1, const char *s2 )`	Copies string `s2` into array `s1`. The value of `s1` is returned.
`char *strncpy( char *s1, const char *s2, size_t n )`	Copies at most `n` characters of string `s2` into array `s1`. The value of `s1` is returned.
`char *strcat( char *s1, const char *s2 )`	Appends string `s2` to array `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.
`char *strncat( char *s1, const char *s2, size_t n )`	Appends at most `n` characters of string `s2` to array `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.

418

```c
/* Fig. 8.18: fig08_18.c
   Using strcpy and strncpy */
#include <stdio.h>
#include <string.h>

int main( void )
{
   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( "%s%s\n%s%s\n",
      "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'; /* terminate string in z */
   printf( "The string in array z is: %s\n", z );

   return 0; /* indicates successful termination */

} /* end main */
```

```
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
```

- fig08_18.c

strcpy copies string **x** into character array **y**

strncpy copies 14 characters of string **x** into character array **z**

Note that **strncpy** does not automatically append a null character

419

```c
/* Fig. 8.19: fig08_19.c
   Using strcat and strncat */
#include <stdio.h>
#include <string.h>

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

   printf( "s1 = %s\ns2 = %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 ) );

```

- fig08_19.c (1 of 2)

strcat adds the characters of string **s2** to the end of string **s1**

strncat adds the first 6 characters of string **s1** to the end of string **s3**. **A terminating null character is automatically appended to the result.**

420

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

   return 0; /* indicates successful termination */

} /* end main */
```

```
s1 = Happy
s2 = New Year
strcat( s1, s2 ) = Happy New Year
strncat( s3, s1, 6 ) = Happy
strcat( s3, s1 ) = Happy Happy New Year
```

- fig08_19.c (2 of 2)

421

OBJECTIVES

- Fundamentals of strings and characters
- Character-handling library
- String-conversion functions
- Standard input/output library functions
- String-manipulation functions of the string-handling library
- ★Comparison functions of the string-handling library
  - Search functions of the string-handling library
  - Other functions of the string-handling library
- Review

422

Comparison Functions of the String Handling Library

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

```
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`

```
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

423

```c
/* Fig. 8.21: fig08_21.c
   Using strcmp and strncmp */
#include <stdio.h>
#include <string.h>

int main( void )
{
   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\n%s%s\n%s%s\n\n%s%2d\n%s%2d\n%s%2d\n\n",
          "s1 = ", s1, "s2 = ", s2, "s3 = ", s3,
          "strcmp(s1, s2) = ", strcmp( s1, s2 ),
          "strcmp(s1, s3) = ", strcmp( s1, s3 ),
          "strcmp(s3, s1) = ", strcmp( s3, s1 ) );

```

strcmp compares string **s1** to string **s2**

- fig08_21.c (1 of 2)

424

```c
   printf("%s%2d\n%s%2d\n%s%2d\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

strcmp(s1, s2) =  0
strcmp(s1, s3) =  1
strcmp(s3, s1) = -1
```

strncmp compares the first 6 characters of string **s1** to the first 6 characters of string **s3**

```
strncmp(s1, s3, 6) =  0
strncmp(s1, s3, 7) =  1
strncmp(s3, s1, 7) = -1
```

- fig08_21.c (2 of 2)

425

OBJECTIVES

- Fundamentals of strings and characters
- Character-handling library
- String-conversion functions
- Standard input/output library functions
- String-manipulation functions of the string-handling library
  - Comparison functions of the string-handling library
  - Search functions of the string-handling library
  - Other functions of the string-handling library
- Review

426

Search Functions of the String-Handling Library

Function prototype	Function description
`char *strchr( const char *s, int c );`	Locates the first occurrence of character `c` in string `s`. If `c` is found, a pointer to `c` in `s` is returned. Otherwise, a `NULL` pointer is returned.
`size_t strcspn( const char *s1, const char *s2 );`	Determines and returns the length of the initial segment of string `s1` consisting of characters not contained in string `s2`.
`size_t strspn( const char *s1, const char *s2 );`	Determines and returns the length of the initial segment of string `s1` consisting only of characters contained in string `s2`.
`char *strpbrk( const char *s1, const char *s2 );`	Locates the first occurrence in string `s1` of any character in string `s2`. If a character from string `s2` is found, a pointer to the character in string `s1` is returned. Otherwise, a `NULL` pointer is returned.

427

Search Functions of the String-Handling Library

Function prototype	Function description
`char *strrchr( const char *s, int c );`	Locates the last occurrence of `c` in string `s`. If `c` is found, a pointer to `c` in string `s` is returned. Otherwise, a `NULL` pointer is returned.
`char *strstr( const char *s1, const char *s2 );`	Locates the first occurrence in string `s1` of string `s2`. If the string is found, a pointer to the string in `s1` is returned. Otherwise, a `NULL` pointer is returned.
`char *strtok( char *s1, const char *s2 );`	A sequence of calls to `strtok` breaks string `s1` into "tokens"—logical pieces such as words in a line of text—separated by characters contained in string `s2`. The first call contains `s1` as the first argument, and subsequent calls to continue tokenizing the same string contain `NULL` 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, `NULL` is returned.

428

```c
/* Fig. 8.23: fig08_23.c
   Using strchr */
#include <stdio.h>
#include <string.h>

int main( void )
{
   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 \"%s\".\n",
         character1, string );
   } /* end if */
   else { /* if character1 was not found */
      printf( "\'%c\' was not found in \"%s\".\n",
         character1, string );
   } /* end else */
```

strchr searches for the first instance of **character1** in **string**

- fig08_23.c (1 of 2)

429

```c

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

   return 0; /* indicates successful termination */

} /* end main */
```

```
'a' was found in "This is a test".
'z' was not found in "This is a test".
```

- fig08_23.c (2 of 2)

430

```c
/* Fig. 8.24: fig08_24.c
   Using strcspn */
#include <stdio.h>
#include <string.h>

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

   printf( "%s%s\n%s%s\n\n%s\n%s%u\n",
      "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 */

} /* end main */
```

- fig08_24.c

strcspn returns the length of the initial segment of **string1** that does not contain any characters in **string2**

```
string1 = The value is 3.14159
string2 = 1234567890

The length of the initial segment of string1
containing no characters from string2 = 13
```

431

```c
/* Fig. 8.27: fig08_27.c
   Using strspn */
#include <stdio.h>
#include <string.h>

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

   printf( "%s%s\n%s%s\n\n%s\n%s%u\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 */

} /* end main */
```

- fig08_27.c

strspn returns the length of the initial segment of **string1** that contains only characters from **string2**

```
string1 = The value is 3.14159
string2 = aehi lsTuv

The length of the initial segment of string1
containing only characters from string2 = 13
```

432

```c
/* Fig. 8.25: fig08_25.c
   Using strpbrk */
#include <stdio.h>
#include <string.h>

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

   printf( "%s\"%s\"\n'%c'%s\n\"%s\"\n",
      "Of the characters in ", string2,
      *strpbrk( string1, string2 ),
      " appears earliest in ", string1 );

   return 0; /* indicates successful termination */

} /* end main */
```

strpbrk returns a pointer to the first appearance in **string1** of any character from **string2**

```
Of the characters in "beware"
'a' appears earliest in
"This is a test"
```

- fig08_25.c

433

```c
/* Fig. 8.26: fig08_26.c
   Using strrchr */
#include <stdio.h>
#include <string.h>

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

   int c = 'z'; /* character to search for */

   printf( "%s\n%s'%c'%s\"%s\"\n",
           "The remainder of string1 beginning with the",
           "last occurrence of character ", c,
           " is: ", strrchr( string1, c ) );

   return 0; /* indicates successful termination */

} /* end main */
```

strrchr returns the remainder of **string1** following the last occurrence of the character **c**

```
The remainder of string1 beginning with the
last occurrence of character 'z' is: "zebras"
```

- fig08_26.c

434

```c
/* Fig. 8.28: fig08_28.c
   Using strstr */
#include <stdio.h>
#include <string.h>

int main( void )
{
   const char *string1 = "abcdefabcdef"; /* string to search */
   const char *string2 = "def"; /* string to search for */

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

   return 0; /* indicates successful termination */

} /* end main */
```

```
string1 = abcdefabcdef
string2 = def

The remainder of string1 beginning with the
first occurrence of string2 is: defabcdef
```

- fig08_28.c

strstr returns the remainder of **string1** following the last occurrence of **string2**

435

```c
/* Fig. 8.29: fig08_29.c
   Using strtok */
#include <stdio.h>
#include <string.h>

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

   printf( "%s\n%s\n\n%s\n",
      "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\n", tokenPtr );
      tokenPtr = strtok( NULL, " " ); /* get next token */
   } /* end while */
```

- fig08_29.c (1 of 2)

strtok "tokenizes" **string** by breaking it into tokens at each space

Calling **strtok** again and passing it **NULL** continues tokenizing from the location in string saved by the last call to **strtok**.

436

```c

   return 0; /* indicates successful termination */

} /* end main */
```

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:
This
is
a
sentence
with
7
tokens
```

- fig08_29.c (2 of 2)

437

OBJECTIVES

- Fundamentals of strings and characters
- Character-handling library
- String-conversion functions
- Standard input/output library functions
- String-manipulation functions of the string-handling library
  - Comparison functions of the string-handling library
  - Search functions of the string-handling library
  - Other functions of the string-handling library
- Review

438

Other Functions of the String Handling Library

Function prototype	Function description
`char *strerror( int errornum );`	Maps `errornum` into a full text string in a locale-specific manner (e.g. the message may appear in different languages based on its location). A pointer to the string is returned.
`size_t strlen( const char *s );`	Determines the length of string `s`. The number of characters preceding the terminating null character is returned.

439

```c
/* Fig. 8.37: fig08_37.c
   Using strerror */
#include <stdio.h>
#include <string.h>

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

   return 0; /* indicates successful termination */

} /* end main */
```

`strerror` returns an error message based on the number passed to it

```
No such file or directory
```

- fig08_37.c

440

```c
/* Fig. 8.38: fig08_38.c
   Using strlen */
#include <stdio.h>
#include <string.h>

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

   printf("%s\"%s\"%s%lu\n%s\"%s\"%s%lu\n%s\"%s\"%s%lu\n",
      "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 */

} /* end main */
```

strlen returns the length of **string1**

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

- fig08_38.c

441

OBJECTIVES

- Fundamentals of strings and characters
- Character-handling library
- String-conversion functions
- Standard input/output library functions
- String-manipulation functions of the string-handling library
- Comparison functions of the string-handling library
- Search functions of the string-handling library
- Other functions of the string-handling library
- Review

442

Review

- There are a lot of string manipulation functions in this chapter. Theses functions are very useful in many practical problems.
- The best way to understand this chapter is to use these functions more.
- In the following tutorial, you are asked to repeat most programs discussed in this chapter in order to grasp how to use these functions.

443

The End

- Thank you very much!

444

Chapter 9

Formatted Input/Output

Acknowledgment

The notes are adapted from those provided by Deitel & Associates, Inc. and Pearson Education Inc.

445

OBJECTIVES

- Streams
- Formatting output with printf
- To use all input formatting capabilities
- To print with field widths and precisions
- To use formatting flags in the printf format control string
- Printing literals and escape sequences
- Reading formatted input with scanf

446

Introduction

- In this chapter
  - Presentation of results
  - `scanf` and `printf`
  - Streams (input and output)
    - `gets`, `puts`, `getchar`, `putchar` (in `<stdio.h>`)

447

OBJECTIVES

- Streams
- Formatting output with printf
- To use all input formatting capabilities
- To print with field widths and precisions
- To use formatting flags in the printf format control string
- Printing literals and escape sequences
- Reading formatted input with scanf

448

Streams

- Streams
  - Sequences of characters organized into lines
    - Each line consists of zero or more characters and ends with newline character
    - ANSI C must support lines of at least 254 characters
  - Performs all input and output
  - Can often be redirected
    - Standard input – keyboard
    - Standard output – screen
    - Standard error – screen

449

OBJECTIVES

- Streams
- Formatting output with printf
- To use all input formatting capabilities
- To print with field widths and precisions
- To use formatting flags in the printf format control string
- Printing literals and escape sequences
- Reading formatted input with scanf

450

Formatting Output with `printf`

- `printf`
  - Precise output formatting
    - Conversion specifications: flags, field widths, precisions, etc.
  - Can perform rounding, aligning columns, right/left justification, inserting literal characters, exponential format, hexadecimal format, and fixed width and precision
- Format
  - `printf(` *format-control-string, other-arguments* `);`
  - Format control string: describes output format
  - Other-arguments: correspond to each conversion specification in format-control-string
    - Each specification begins with a percent sign(%), ends with conversion specifier

451

OBJECTIVES

- Streams
- Formatting output with printf
- To use all input formatting capabilities
- To print with field widths and precisions
- To use formatting flags in the printf format control string
- Printing literals and escape sequences
- Reading formatted input with scanf

452

Printing Integers

- Integer
  - Whole number (no decimal point): `25, 0, -9`
  - Positive, negative, or zero
  - Only minus sign prints by default (later we shall change this)

453

Integer conversion specifiers

Conversion specifier	Description
`d`	Display as a signed decimal integer.
`i`	Display as a signed decimal integer. [*Note:* The `i` and `d` specifiers are different when used with `scanf`.]
`o`	Display as an unsigned octal integer.
`u`	Display as an unsigned decimal integer.
`x` or `X`	Display as an unsigned hexadecimal integer. `X` causes the digits `0-9` and the letters `A-F` to be displayed and `x` causes the digits `0-9` and `a-f` to be displayed.
`h` or `l` (letter `l`)	Place before any integer conversion specifier to indicate that a `short` or `long` integer is displayed, respectively. Letters `h` and `l` are more precisely called **length modifiers**.

454

```c
/* Fig 9.2: fig09_02.c */
/* Using the integer conversion specifiers */
#include <stdio.h>

int main( void )
{
   printf( "%d\n", 455 );
   printf( "%i\n", 455 );  /* i same as d in printf */
   printf( "%d\n", +455 );
   printf( "%d\n", -455 );
   printf( "%hd\n", 32000 );
   printf( "%ld\n", 2000000000L ); /* L suffix makes literal a long */
   printf( "%o\n", 455 );
   printf( "%u\n", 455 );
```

d and **i** specify signed integers

h specifies a **short** number

l specifies a **long** number

o specifies an octal integer

u specifies an unsigned integer

- fig09_02.c (1 of 2)

455

```c
15    printf( "%u\n", -455 );
16    printf( "%x\n", 455 );
17    printf( "%X\n", 455 );
18
19    return 0; /* indicates successful termination */
20
21 } /* end main */
```

x and **X** specify hexadecimal integers

```
455
455
455
-455
32000
2000000000
707
455
4294966841
1c7
1C7
```

Note: The value -455, when read by %u (line 15), is converted to the unsigned value 4294966841. Hence, **Printing a negative value with a conversion specifier that expects an unsigned value is an error.**

- fig09_02.c (2 of 2)

456

Printing Floating-Point Numbers

- Floating Point Numbers
  - Have a decimal point (`33.5`)
  - Exponential notation (computer's version of scientific notation)
    - `150.3` is `1.503 x 10²` in scientific
    - `150.3` is `1.503E+02` in exponential (`E` stands for exponent)
    - use `e` or `E`
  - `f` – print floating point with at least one digit to left of decimal
  - `g` (or `G`) - prints in `f` or `e` with no trailing zeros (`1.2300` becomes `1.23`)
    - Use exponential if exponent less than `-4`, or greater than or equal to precision (`6` digits by default)

457

Floating-point conversion specifiers

Conversion specifier	Description
`e` or `E`	Display a floating-point value in exponential notation.
`f`	Display floating-point values in fixed-point notation.
`g` or `G`	Display a floating-point value in either the floating-point form `f` or the exponential form `e` (or `E`), based on the magnitude of the value.
`L`	Place before any floating-point conversion specifier to indicate that a `long double` floating-point value is displayed.

458

```c
/* Fig 9.4: fig09_04.c */
/* Printing floating-point numbers with
   floating-point conversion specifiers */

#include <stdio.h>

int main( void )
{
   printf( "%e\n", 1234567.89 );
   printf( "%e\n", +1234567.89 );
   printf( "%e\n", -1234567.89 );
   printf( "%E\n", 1234567.89 );
   printf( "%f\n", 1234567.89 );
   printf( "%g\n", 1234567.89 );
   printf( "%G\n", 1234567.89 );

   return 0; /* indicates successful termination */

} /* end main */
```

e and **E** specify exponential notation

f specifies fixed-point notation

g and **G** specify either exponential or fixed-point notation depending on the number's size

```
1.234568e+006
1.234568e+006
-1.234568e+006
1.234568E+006
1234567.890000
1.23457e+006
1.23457E+006
```

- fig09_04.c

459

Printing Strings and Characters

- `c`
  - Prints `char` argument
  - Cannot be used to print the first character of a string
- `s`
  - Requires a pointer to `char` as an argument
  - Prints characters until `NULL` (`'\0'`) encountered
  - Cannot print a `char` argument
- Remember
  - Single quotes for character constants (`'z'`)
  - Double quotes for strings `"z"` (which actually contains two characters, `'z'` and `'\0'`)

460

```c
/* Fig 9.5: fig09_05c */
/* Printing strings and characters */
#include <stdio.h>

int main( void )
{
   char character = 'A'; /* initialize char */
   char string[] = "This is a string"; /* initialize char array */
   const char *stringPtr = "This is also a string"; /* char pointer */

   printf( "%c\n", character );
   printf( "%s\n", "This is a string" );
   printf( "%s\n", string );
   printf( "%s\n", stringPtr );

   return 0; /* indicates successful termination */

} /* end main */
```

c specifies a character will be printed

s specifies a string will be printed

```
A
This is a string
This is a string
This is also a string
```

- fig09_05.c

461

Other Conversion Specifiers

- p
  - Displays pointer value (i.e. address) in an implementation-defined manner (one many systems, hexadecimal notation is used rather than decimal notation).
- n
  - Stores number of characters already output by current `printf` statement
  - Takes a pointer to an integer as an argument
  - Nothing printed by a %n specification
  - Every `printf` call returns a value
    - Number of characters output
    - Negative number if error occurs
- %
  - Prints a percent sign
  - %%

462

```c
/* Fig 9.7: fig09_07.c */
/* Using the p, n, and % conversion specifiers */
#include <stdio.h>

int main( void )
{
   int *ptr;      /* define pointer to int */
   int x = 12345; /* initialize int x */
   int y;         /* define int y */

   ptr = &x;      /* assign address of x to ptr */
   printf( "The value of ptr is %p\n", ptr );
   printf( "The address of x is %p\n\n", &x );

   printf( "Total characters printed on this line:%n", &y );
   printf( " %d\n\n", y );

   y = printf( "This line has 28 characters\n%n" );
   printf( "%d characters were printed\n\n", y );

   printf( "Printing a %% in a format control string\n" );
```

- fig09_07.c (1 of 2)

p specifies a memory address will be printed

n stores the number of characters printed on a line

% prints a percent sign

463

```c

   return 0; /* indicates successful termination */

} /* end main */
```

```
The value of ptr is 0012FF78
The address of x is 0012FF78

Total characters printed on this line: 38

This line has 28 characters
28 characters were printed

Printing a % in a format control string
```

- fig09_07.c (2 of 2)

464

Printing with Field Widths and Precision

- Field width
  - Size of field in which data is printed
  - If width larger than data, right justified by default
    - If field width too small, increases to fit data
    - Minus sign uses one character position in field
  - Integer width inserted between % and conversion specifier
  - %4d – field width of 4

465

```c
/* Fig 9.8: fig09_08.c */
/* Printing integers right-justified */
#include <stdio.h>

int main( void )
{
   printf( "%4d\n", 1 );
   printf( "%4d\n", 12 );
   printf( "%4d\n", 123 );
   printf( "%4d\n", 1234 );
   printf( "%4d\n\n", 12345 );

   printf( "%4d\n", -1 );
   printf( "%4d\n", -12 );
```

A field width of 4 will make C attempt to print the number in a 4-character space

Note that C considers the minus sign a character

- fig09_08.c (1 of 2)

466

```c
   printf( "%4d\n", -123 );
   printf( "%4d\n", -1234 );
   printf( "%4d\n", -12345 );

   return 0; /* indicates successful termination */

} /* end main */
```

```
   1
  12
 123
1234
12345

  -1
 -12
-123
-1234
-12345
```

- fig09_08.c (2 of 2)

467

Printing with Field Widths and Precision

- Precision
  - Meaning varies depending on data type
  - Integers (default 1)
    - Minimum number of digits to print
      - If data too small, prefixed with zeros
  - Floating point
    - Number of digits to appear after decimal (e and f)
      - For g – maximum number of significant digits
    - When a floating-point value is printed with a precision smaller than the original number of decimal places in the value, the value is rounded.
  - Strings
    - Maximum number of characters to be written from string
  - Format
    - Use a dot (.) then precision number after %
      %.3f

468

OBJECTIVES

- Streams
- Formatting output with printf
- To use all input formatting capabilities
- To print with field widths and precisions
- To use formatting flags in the printf format control string
- Printing literals and escape sequences
- Reading formatted input with scanf

469

Printing with Field Widths and Precisions

- Field width and precision
  - Can both be specified
    - `%width.precision`
      `%5.3f`
  - Negative field width – left justified
  - Positive field width – right justified
  - Precision must be positive
  - Can use integer expressions to determine field width and precision values
    - Place an asterisk (`*`) in place of the field width or precision
      - Matched to an `int` argument in argument list
    - Example:
      `printf( "%*.*f", 7, 2, 98.736 );`

470

```c
/* Fig 9.9: fig09_09.c */
/* Using precision while printing integers,
   floating-point numbers, and strings */
#include <stdio.h>

int main( void )
{
   int i = 873;                  /* initialize int i */
   double f = 123.94536;         /* initialize double f */
   char s[] = "Happy Birthday"; /* initialize char array s */

   printf( "Using precision for integers\n" );
   printf( "\t%.4d\n\t%.9d\n\n", i, i );

   printf( "Using precision for floating-point numbers\n" );
   printf( "\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f );
```

Precision for integers specifies the minimum number of characters to be printed

Precision for the **g** specifier controls the maximum number of significant digits printed

Precision for **f** and **e** specifiers controls the number of digits after the decimal point

- fig09_09.c (1 of 2)


```c

   printf( "Using precision for strings\n" );
   printf( "\t%.11s\n", s );

   return 0; /* indicates successful termination */

} /* end main */
```

```
Using precision for integers
        0873
        000000873

Using precision for floating-point numbers
        123.945
        1.239e+002
        124

Using precision for strings
        Happy Birth
```

Precision for strings specifies the maximum number of characters to be printed

- fig09_09.c (2 of 2)

OBJECTIVES

- Streams
- Formatting output with printf
- To use all input formatting capabilities
- To print with field widths and precisions
- To use formatting flags in the printf format control string
- Printing literals and escape sequences
- Reading formatted input with scanf

473

Using Flags in the `printf` Format-Control String

- Flags
  - Supplement formatting capabilities
  - Place flag immediately to the right of percent sign
  - Several flags may be combined

Flag	Description
- (minus sign)	Left justify the output within the specified field.
+ (plus sign)	Display a plus sign preceding positive values and a minus sign preceding negative values.
space	Print a space before a positive value not printed with the + flag.
#	Prefix 0 to the output value when used with the octal conversion specifier o.
	Prefix 0x or 0X to the output value when used with the hexadecimal conversion specifiers x or X.
	Force a decimal point for a floating-point number printed with e, E, f, g or G that does not contain a fractional part. (Normally the decimal point is printed only if a digit follows it.) For g and G specifiers, trailing zeros are not eliminated.
0 (zero)	Pad a field with leading zeros.

Format control string flags

474

```c
/* Fig 9.11: fig09_11.c */
/* Right justifying and left justifying values */
#include <stdio.h>

int main( void )
{
   printf( "%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23 );
   printf( "%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23 );

   return 0; /* indicates successful termination */

} /* end main */
```

– flag left justifies characters in a field

```
     hello         7         a  1.230000

hello     7         a         1.230000
```

- fig09_11.c

475

```c
/* Fig 9.12: fig09_12.c */
/* Printing numbers with and without the + flag */
#include <stdio.h>

int main( void )
{
   printf( "%d\n%d\n", 786, -786 );
   printf( "%+d\n%+d\n", 786, -786 );

   return 0; /* indicates successful termination */

} /* end main */
```

+ flag forces a plus sign on positive numbers

```
786
-786
+786
-786
```

- fig09_12.c

476

```c
/* Fig 9.13: fig09_13.c */
/* Printing a space before signed values
   not preceded by + or - */
#include <stdio.h>

int main( void )
{
   printf( "% d\n% d\n", 547, -547 );

   return 0; /* indicates successful termination */

} /* end main */
```

Space flag forces a space on positive numbers

```
 547
-547
```

- fig09_13.c

477

```c
/* Fig 9.14: fig09_14.c */
/* Using the # flag with conversion specifiers
   o, x, X and any floating-point specifier */
#include <stdio.h>

int main( void )
{
   int c = 1427;      /* initialize c */
   double p = 1427.0; /* initialize p */

   printf( "%#o\n", c );
   printf( "%#x\n", c );
   printf( "%#X\n", c );
   printf( "\n%g\n", p );
   printf( "%#g\n", p );

   return 0; /* indicates successful termination */

} /* end main */
```

flag prefixes a **0** before octal integers

flag prefixes a **0x** before hexadecimal integers

flag forces a decimal point on floating-point numbers with no fractional part

```
02623
0x593
0X593

1427
1427.00
```

- fig09_14.c

478

```c
/* Fig 9.15: fig09_15.c */
/* Printing with the 0( zero ) flag fills in leading zeros */
#include <stdio.h>

int main( void )
{
   printf( "%+09d\n", 452 );
   printf( "%09d\n", 452 );

   return 0; /* indicates successful termination */

} /* end main */
```

0 flag fills empty spaces with zeros

```
+00000452
000000452
```

- fig09_15.c

479

OBJECTIVES

- Streams
- Formatting output with printf
- To use all input formatting capabilities
- To print with field widths and precisions
- To use formatting flags in the printf format control string
- Printing literals and escape sequences
- Reading formatted input with scanf

480

Printing Literals and Escape Sequences

- Printing Literals
  - Most characters can be printed
  - Certain "problem" characters, such as the quotation mark "
  - Must be represented by escape sequences
    - Represented by a backslash \ followed by an escape character

481

Escape sequences

Escape sequence	Description
`\'` (single quote)	Output the single quote (`'`) character.
`\"` (double quote)	Output the double quote (`"`) character.
`\?` (question mark)	Output the question mark (`?`) character.
`\\` (backslash)	Output the backslash (`\`) character.
`\a` (alert or bell)	Cause an audible (bell) or visual alert.
`\b` (backspace)	Move the cursor back one position on the current line.
`\f` (new page or form feed)	Move the cursor to the start of the next logical page.
`\n` (newline)	Move the cursor to the beginning of the next line.
`\r` (carriage return)	Move the cursor to the beginning of the current line.
`\t` (horizontal tab)	Move the cursor to the next horizontal tab position.
`\v` (vertical tab)	Move the cursor to the next vertical tab position.

482

OBJECTIVES

- Streams
- Formatting output with printf
- To use all input formatting capabilities
- To print with field widths and precisions
- To use formatting flags in the printf format control string
- Printing literals and escape sequences
- Reading formatted input with scanf

483

Formatting Input with `scanf`

- `scanf`
  - Input can be formatted much like output can
  - `scanf` conversion specifiers are slightly different from those used with `printf`

484

Conversion specifiers for `scanf`. (Part 1 of 2.)

Conversion specifier	Description
Integers	
`d`	Read an optionally signed decimal integer. The corresponding argument is a pointer to an `int`.
`i`	Read an optionally signed decimal, octal or hexadecimal integer. The corresponding argument is a pointer to an `int`.
`o`	Read an octal integer. The corresponding argument is a pointer to an unsigned `int`.
`u`	Read an unsigned decimal integer. The corresponding argument is a pointer to an unsigned `int`.
`x` or `X`	Read a hexadecimal integer. The corresponding argument is a pointer to an unsigned `int`.
`h` or `l`	Place before any of the integer conversion specifiers to indicate that a `short` or `long` integer is to be input.

485

Conversion specifiers for `scanf`. (Part 2 of 2.)

Conversion specifier	Description
Floating-point numbers	
`e`, `E`, `f`, `g` or `G`	Read a floating-point value. The corresponding argument is a pointer to a floating-point variable.
`l` or `L`	Place before any of the floating-point conversion specifiers to indicate that a `double` or `long double` value is to be input. The corresponding argument is a pointer to a `double` or `long double` variable.
Characters and strings	
`c`	Read a character. The corresponding argument is a pointer to a `char`; no null `('\0')` is added.
`s`	Read a string. The corresponding argument is a pointer to an array of type `char` that is large enough to hold the string and a terminating null `('\0')` character—which is automatically added.

486

```c
/* Fig 9.18: fig09_18.c */
/* Reading integers */
#include <stdio.h>

int main( void )
{
   int a;
   int b;
   int c;
   int d;
   int e;
   int f;
   int g;

   printf( "Enter seven integers: " );
   scanf( "%d%i%i%o%u%x", &a, &b, &c, &d, &e, &f, &g );

   printf( "The input displayed as decimal integers is:\n" );
   printf( "%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g );

   return 0; /* indicates successful termination */

} /* end main */
```

d specifies a decimal integer will be input

i specifies an integer will be input

o specifies an octal integer will be input

u specifies an unsigned decimal integer will be input

x specifies a hexadecimal integer will be input

```
Enter seven integers: -70 -70 070 0x70 70 70 70
The input displayed as decimal integers is:
-70 -70 56 112 56 70 112
```

- fig09_18.c

487

```c
/* Fig 9.19: fig09_19.c */
/* Reading floating-point numbers */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   double a;
   double b;
   double c;

   printf( "Enter three floating-point numbers: \n" );
   scanf( "%le%lf%lg", &a, &b, &c );

   printf( "Here are the numbers entered in plain\n" );
   printf( "floating-point notation:\n" );
   printf( "%f\n%f\n%f\n", a, b, c );

   return 0; /* indicates successful termination */

} /* end main */
```

e, **f**, and **g** specify a floating-point number will be input

l specifies a **double** or **long double** will be input

```
Enter three floating-point numbers:
1.27987 1.27987e+03 3.38476e-06
Here are the numbers entered in plain
floating-point notation:
1.279870
1279.870000
0.000003
```

- fig09_19.c

488

```c
/* Fig 9.20: fig09_20.c */
/* Reading characters and strings */
#include <stdio.h>

int main( void )
{
   char x;
   char y[ 9 ];

   printf( "Enter a string: " );
   scanf( "%c%s", &x, y );

   printf( "The input was:\n" );
   printf( "the character \"%c\" ", x );
   printf( "and the string \"%s\"\n", y );

   return 0; /* indicates successful termination */

} /* end main */
```

c specifies a character will be input

s specifies a string will be input

```
Enter a string: Sunday
The input was:
the character "S" and the string "unday"
```

- fig09_20.c

489

```c
/* Fig 9.21: fig09_21.c */
/* Using a scan set */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   char z[ 9 ]; /* define array z */

   printf( "Enter string: " );
   scanf( "%[aeiou]", z ); /* search for set of characters */

   printf( "The input was \"%s\"\n", z );

   return 0; /* indicates successful termination */

} /* end main */
```

- fig09_21.c

[] specifies only the initial segment of a string that contains the characters in brackets will be read

```
Enter string: ooeeooahah
The input was "ooeeooa"
```

Note: A scan set is a set of characters enclosed in []. The scan set stops inputting characters when a character that is not contained in the scan set is encountered. If the first character in the input stream does not match a character in the scan set, only the null character is stored in the array.

490

```c
/* Fig 9.22: fig09_22.c */
/* Using an inverted scan set */
#include <stdio.h>

int main( void )
{
   char z[ 9 ];

   printf( "Enter a string: " );
   scanf( "%[^aeiou]", z ); /* inverted scan set */

   printf( "The input was \"%s\"\n", z );

   return 0; /* indicates successful termination */

} /* end main */
```

```
Enter a string: String
The input was "Str"
```

[] and ^ specify only the initial segment of a string that does **not** contain the characters in brackets will be read

Note: The scan set can also be used to scan for characters not contained in the scan set by using an inverted scan set ^[].

- fig09_22.c

491

```c
/* Fig 9.23: fig09_23.c */
/* inputting data with a field width */
#include <stdio.h>

int main( void )
{
   int x;
   int y;

   printf( "Enter a six digit integer: " );
   scanf( "%2d%d", &x, &y );

   printf( "The integers input were %d and %d\n", x, y );

   return 0; /* indicates successful termination */

} /* end main */
```

```
Enter a six digit integer: 123456
The integers input were 12 and 3456
```

A field width of 2 tells C to only read the first 2 characters of that input

- fig09_23.c

492

```c
/* Fig 9.24: fig09_24.c */
/* Reading and discarding characters from the input stream */
#include <stdio.h>

int main( void )
{
   int month1;
   int day1;
   int year1;
   int month2;
   int day2;
   int year2;

   printf( "Enter a date in the form mm-dd-yyyy: " );
   scanf( "%d%*c%d%*c%d", &month1, &day1, &year1 );

   printf( "month = %d  day = %d  year = %d\n\n", month1, day1, year1 );

   printf( "Enter a date in the form mm/dd/yyyy: " );
   scanf( "%d%*c%d%*c%d", &month2, &day2, &year2 );

   printf( "month = %d  day = %d  year = %d\n", month2, day2, year2 );

   return 0; /* indicates successful termination */

} /* end main */
```

* is a wildcard—**scanf** will disregard anything between the two inputs on either side of it

```
Enter a date in the form mm-dd-yyyy: 11-18-2003
month = 11  day = 18  year = 2003

Enter a date in the form mm/dd/yyyy: 11/18/2003
month = 11  day = 18  year = 2003
```

- fig09_24.c

493

Review

- Printf ();
  - Format control: %d %ld %f %e %g %c %s
  - Fieldwidth.precision
  - Escape sequence %% /n /t
- Scanf();
  - Scanf("%10s", string);

494

The End

- Thank you very much!

495

Chapter 10

C Structures, Unions, Bit Manipulations and Enumerations

Acknowledgment

The notes are adapted from those provided by Deitel & Associates, Inc. and Pearson Education Inc.

496

OBJECTIVES

- To create, initialize and use structures
- To pass structures to functions by value and by reference (Typedef)
- Example: Card shuffling and dealing
- To create, initialize and use unions
- Bitwise operators and bit fields
- To create, initialize and use enumeration

497

Structure Definitions

- Example

```
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`

498

Structure Definitions

- `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;
```

499

Structure Definitions

- 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

500

Note:

1. Structure members are not necessarily stored in consecutive bytes of memory. Sometimes there are "holes" in a structure, because computers may store specific data types only on certain memory boundaries such as **half word**, **word** or **double word boundaries**. A word is a standard memory unit used to store data in a computer, usually 2 bytes or 4 bytes.

e.g.

```
struct example {
        char c;
        int i;
} sample1, sample2;
```

Even if the member values of sample1 and sample2 are equal, the structures are not necessarily equal, because the undefined 1-byte holes are not likely to contain identical values.

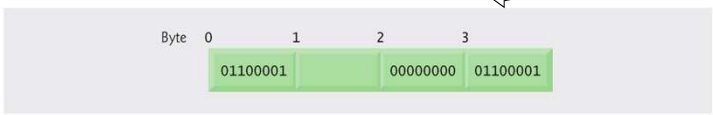


Fig. 10.1 | Possible storage alignment for a variable of type `struct` example showing an undefined area in memory, where the variable has been assigned the character 'a' and the integer 97.

501

Initializing Structures

- Initializer lists
  - Example:

```
struct card oneCard = { "Three", "Hearts"
  };
```

- Assignment statements
  - Example:

```
struct card threeHearts = oneCard;
```

  - Could also define and initialize `threeHearts` as follows:

```
structure card threeHearts;
threeHearts.face = "Three";
threeHearts.suit = "Hearts";
```

502

Accessing Members of Structures

- Accessing structure members
  - Dot operator (`.`) used with structure variables

    ```
    struct card myCard;
    printf( "%s", myCard.suit );
    ```
  - Arrow operator (`->`) used with pointers to structure variables

    ```
    struct card *myCardPtr = &myCard;
    printf( "%s", myCardPtr->suit );
    ```
  - `myCardPtr->suit` is equivalent to `( *myCardPtr ).suit`

503

```
/* 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( void )
{
   struct card aCard; /* define one struct card variable */
   struct card *cardPtr; /* define a pointer to a struct card */

   /* place strings into aCard */
   aCard.face = "Ace";
   aCard.suit = "Spades";
```

Structure definition

Structure definition must end with semicolon

Dot operator accesses members of a structure

- fig10_02.c (1 of 2)

504

```c

   cardPtr = &aCard; /* assign address of aCard to cardPtr */

   printf( "%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
      cardPtr->face, " of ", cardPtr->suit,
      ( *cardPtr ).face, " of ", ( *cardPtr ).suit );

   return 0; /* indicates successful termination */

} /* end main */
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

Arrow operator accesses members of a structure pointer

- fig10_02.c (2 of 2)

505

OBJECTIVES

- To create, initialize and use structures
- To pass structures to functions by value and by reference (Typedef)
- Example: Card shuffling and dealing
- To create, initialize and use unions
- Bitwise operators and bit fields
- To create, initialize and use enumeration

506

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

507

`typedef`

- `typedef`
  - Creates synonyms (aliases) for previously defined data types
  - Use `typedef` to create shorter type names
  - Example:

```
typedef struct Card *CardPtr;
```

  - Defines a new type name `CardPtr` as a synonym for type `struct Card *`
  - `typedef` does not create a new data type
    - Only creates an alias of an existing type name

508

OBJECTIVES

- To create, initialize and use structures
- To pass structures to functions by value and by reference (Typedef)
- Example: Card shuffling and dealing
- To create, initialize and use unions
- Bitwise operators and bit fields
- To create, initialize and use enumeration

509

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

510

```c
/* 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; /* new type name for struct 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( void )
{
   Card deck[ 52 ]; /* define array of Cards */

   /* initialize array of pointers */
   const char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
      "Six", "Seven", "Eight", "Nine", "Ten",
      "Jack", "Queen", "King"};

```

Each **card** has a face and a suit

Card is now an alias for **struct card**

- fig10_03.c (1 of 3)

511

```c
   /* 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 */

```

Constant pointer to modifiable array of **Card**s

Fills the deck by giving each **Card** a face and suit

- fig10_03.c (2 of 3)

512

```c
/* 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 */

/* deal cards */
void deal( const Card * const wDeck )
{
   int i; /* counter */

   /* loop through wDeck */
   for ( i = 0; i <= 51; i++ ) {
      printf( "%5s of %-8s%c", wDeck[ i ].face, wDeck[ i ].suit,
         ( i + 1 ) % 2 ? '\t' : '\n' );
   } /* end for */

} /* end function deal */
```

Each card is swapped with another, random card, shuffling the deck

- fig10_03.c (3 of 3)

513

```
Four of Clubs          Three of Hearts
Three of Diamonds       Three of Spades
 Four of Diamonds         Ace of Diamonds
 Nine of Hearts           Ten of Clubs
Three of Clubs           Four of Hearts
Eight of Clubs           Nine of Diamonds
Deuce of Clubs          Queen of Clubs
Seven of Clubs           Jack of Spades
  Ace of Clubs           Five of Diamonds
  Ace of Spades          Five of Clubs
Seven of Diamonds         Six of Spades
Eight of Spades         Queen of Hearts
 Five of Spades         Deuce of Diamonds
Queen of Spades           Six of Hearts
Queen of Diamonds       Seven of Hearts
 Jack of Diamonds         Nine of Spades
Eight of Hearts          Five of Hearts
 King of Spades           Six of Clubs
Eight of Diamonds         Ten of Spades
  Ace of Hearts          King of Hearts
 Four of Spades          Jack of Hearts
Deuce of Hearts          Jack of Clubs
Deuce of Spades           Ten of Diamonds
Seven of Spades          Nine of Clubs
 King of Clubs            Six of Diamonds
  Ten of Hearts          King of Diamonds
```

514

OBJECTIVES

- To create, initialize and use structures
- To pass structures to functions by value and by reference (Typedef)
- Example: Card shuffling and dealing
- To create, initialize and use unions
- Bitwise operators and bit fields
- To create, initialize and use enumeration

515

Unions

- `union`
  - Members share the same storage space
  - The members of a union can be of any data type
  - The number of bytes used to store a union must be at least enough to hold the largest member
  - Only one member, and thus one data type, can be referenced at a time
- `union` definitions
  - Same as struct, e.g.

```
union Number {
  int x;
  float y;
};
union Number value;
```

516

Unions

- Valid `union` operations
  - Assignment to union of same type: =
  - Taking address: &
  - Accessing union members: .
  - Accessing members using pointers: ->
- In a declaration, a union my be initialized with a value of the same type as the first union member, e.g.

  union number value = {10};

  union number value = {1.43}

  would truncate the floating-point part of the initializer value and normally would produce a warning from the compiler.

517

```c
/* Fig. 10.5: fig10_05.c
   An example of a union */
#include <stdio.h>

/* number union definition */
union number {
   int x;
   double y;
};/* end union number */

int main( void )
{
   union number value; /* define union variable */

   value.x = 100; /* put an integer into the union */
   printf( "%s\n%s\n%s\n  %d\n\n%s\n  %f\n\n\n",
      "Put a value in the integer member",
      "and print both members.",
      "int:", value.x,
      "double:", value.y );
```

Union definition

Union definition must end with semicolon

Note that **y** has no value

- fig10_05.c (1 of 2)

518

```c
   value.y = 100.0; /* put a double into the same union */
   printf( "%s\n%s\n%s\n  %d\n\n%s\n  %f\n",
      "Put a value in the floating member",
      "and print both members.",
      "int:", value.x,
      "double:", value.y );

   return 0; /* indicates successful termination */

} /* end main */
```

Giving **y** a value removes **x**'s value

```
Put a value in the integer member
and print both members.
int:
  100

double:
  -9255959211743313600000000000000000000000000000000000000000000000.000000


Put a value in the floating member
and print both members.
int:
  0

double:
  100.000000
```

- fig10_05.c (2 of 2)

519

OBJECTIVES

- To create, initialize and use structures
- To pass structures to functions by value and by reference (Typedef)
- Example: Card shuffling and dealing
- To create, initialize and use unions
- Bitwise operators and bit fields
- To create, initialize and use enumeration

520

Bitwise Operators

- All data is represented internally as sequences of bits
  - Each bit can be either 0 or 1
  - Sequence of 8 bits forms a byte
- Bitwise operators are used to manipulate the bits of integral operands
- Bitwise data manipulations are machine dependent
- Unsigned integers are normally used with the bitwise operators

521

Bitwise operators

Operator		Description
&	bitwise AND	The bits in the result are set to 1 if the corresponding bits in the two operands are both 1 .
|	bitwise inclusive OR	The bits in the result are set to 1 if at least one of the corresponding bits in the two operands is 1.
^	bitwise exclusive OR	The bits in the result are set to 1 if exactly one of the corresponding bits in the two operands is 1.
<<	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from the right with 0 bits.
>>	right shift	Shifts the bits of the first operand right by the number of bits specified by the second operand; the method of filling from the left is machine dependent.
~	one's complement	All 0 bits are set to 1 and all 1 bits are set to 0.

522

```c
/* Fig. 10.7: fig10_07.c
   Printing an unsigned integer in bits */
#include <stdio.h>

void displayBits( unsigned value ); /* prototype */

int main( void )
{
   unsigned x; /* variable to hold user input */

   printf( "Enter an unsigned integer: " );
   scanf( "%u", &x );

   displayBits( x );

   return 0; /* indicates successful termination */

} /* end main */

```

- fig10_07.c (1 of 2)

523

```c
/* display bits of an unsigned integer value */
void displayBits( unsigned value )
{
   unsigned c; /* counter */

   /* define displayMask and left shift 31 bits */
   unsigned displayMask = 1 << 31;

   printf( "%10u = ", value );

   /* loop through bits */
   for ( c = 1; c <= 32; c++ ) {
      putchar( value & displayMask ? '1' : '0' );
      value <<= 1; /* shift value left by 1 */

      if ( c % 8 == 0 ) { /* output space after 8 bits */
         putchar( ' ' );
      } /* end if */

   } /* end for */

   putchar( '\n' );
} /* end function displayBits */
```

displayMask is a 1 followed by 31 zeros

Bitwise AND returns nonzero if the leftmost bits of **displayMask** and **value** are both 1, since all other bits in **displayMask** are 0s.

```
Enter an unsigned integer: 65000
     65000 = 00000000 00000000 11111101 11101000
```

- fig10_07.c (2 of 2)

524

Results of combining two bits with the bitwise AND operator &.

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

525

Results of combining two bits with the bitwise inclusive OR operator |.

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

526

Results of combining two bits with the bitwise exclusive OR operator ^.

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1
1	1	0

527

```c
/* Fig. 10.9: fig10_09.c
   Using the bitwise AND, bitwise inclusive OR, bitwise
   exclusive OR and bitwise complement operators */
#include <stdio.h>

void displayBits( unsigned value ); /* prototype */

int main( void )
{
   unsigned number1;
   unsigned number2;
   unsigned mask;
   unsigned setBits;

   /* demonstrate bitwise AND (&) */
   number1 = 65535;
   mask = 1;
   printf( "The result of combining the following\n" );
   displayBits( number1 );
   displayBits( mask );
   printf( "using the bitwise AND operator & is\n" );
   displayBits( number1 & mask );

```

Bitwise AND sets each bit in the result to 1 if the corresponding bits in the operands are both 1

- fig10_09.c (1 of 3)

528

```c
   /* demonstrate bitwise inclusive OR (|) */
   number1 = 15;
   setBits = 241;
   printf( "\nThe result of combining the following\n" );
   displayBits( number1 );
   displayBits( setBits );
   printf( "using the bitwise inclusive OR operator | is\n" );
   displayBits( number1 | setBits );

   /* demonstrate bitwise exclusive OR (^) */
   number1 = 139;
   number2 = 199;
   printf( "\nThe result of combining the following\n" );
   displayBits( number1 );
   displayBits( number2 );
   printf( "using the bitwise exclusive OR operator ^ is\n" );
   displayBits( number1 ^ number2 );

   /* demonstrate bitwise complement (~)*/
   number1 = 21845;
   printf( "\nThe one's complement of\n" );
   displayBits( number1 );
   printf( "is\n" );
   displayBits( ~number1 );

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

- fig10_09.c (2 of 3)

Bitwise inclusive OR sets each bit in the result to 1 if at least one of the corresponding bits in the operands is 1

Bitwise exclusive OR sets each bit in the result to 1 if only one of the corresponding bits in the operands is 1

Complement operator sets each bit in the result to 0 if the corresponding bit in the operand is 1 and vice versa

529

```c
/* display bits of an unsigned integer value */
void displayBits( unsigned value )
{
   unsigned c; /* counter */

   /* declare displayMask and left shift 31 bits */
   unsigned displayMask = 1 << 31;

   printf( "%10u = ", value );

   /* loop through bits */
   for ( c = 1; c <= 32; c++ ) {
      putchar( value & displayMask ? '1' : '0' );
      value <<= 1; /* shift value left by 1 */

      if ( c % 8 == 0 ) { /* output a space after 8 bits */
         putchar( ' ' );
      } /* end if */

   } /* end for */

   putchar( '\n' );
} /* end function displayBits */
```

- fig10_09.c (3 of 3)

530

```
The result of combining the following
     65535 = 00000000 00000000 11111111 11111111
         1 = 00000000 00000000 00000000 00000001
using the bitwise AND operator & is
         1 = 00000000 00000000 00000000 00000001

The result of combining the following
        15 = 00000000 00000000 00000000 00001111
       241 = 00000000 00000000 00000000 11110001
using the bitwise inclusive OR operator | is
       255 = 00000000 00000000 00000000 11111111

The result of combining the following
       139 = 00000000 00000000 00000000 10001011
       199 = 00000000 00000000 00000000 11000111
using the bitwise exclusive OR operator ^ is
        76 = 00000000 00000000 00000000 01001100

The one's complement of
     21845 = 00000000 00000000 01010101 01010101
is
4294945450 = 11111111 11111111 10101010 10101010
```

- fig10_10.c

531

```c
/* Fig. 10.13: fig10_13.c
   Using the bitwise shift operators */
#include <stdio.h>

void displayBits( unsigned value ); /* prototype */

int main( void )
{
   unsigned number1 = 960; /* initialize number1 */

   /* demonstrate bitwise left shift */
   printf( "\nThe result of left shifting\n" );
   displayBits( number1 );
   printf( "8 bit positions using the " );
   printf( "left shift operator << is\n" );
   displayBits( number1 << 8 );

```

Left shift operator shifts all bits left a specified number of spaces, filling in zeros for the empty bits

- fig10_13.c (1 of 3)

532

```c
   /* demonstrate bitwise right shift */
   printf( "\nThe result of right shifting\n" );
   displayBits( number1 );
   printf( "8 bit positions using the " );
   printf( "right shift operator >> is\n" );
   displayBits( number1 >> 8 );

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

/* display bits of an unsigned integer value */
void displayBits( unsigned value )
{
{
   unsigned c; /* counter */

   /* declare displayMask and left shift 31 bits */
   unsigned displayMask = 1 << 31;

   printf( "%7u = ", value );

```

Right shift operator shifts all bits right a specified number of spaces, filling in the empty bits in an implementation-defined manner

- fig10_13.c (2 of 3)

533

```c
   /* loop through bits */
   for ( c = 1; c <= 32; c++ ) {
      putchar( value & displayMask ? '1' : '0' );
      value <<= 1; /* shift value left by 1 */

      if ( c % 8 == 0 ) { /* output a space after 8 bits */
         putchar( ' ' );
      } /* end if */

   } /* end for */

   putchar( '\n' );
} /* end function displayBits */
```

```
The result of left shifting
    960 = 00000000 00000000 00000011 11000000
8 bit positions using the left shift operator << is
 245760 = 00000000 00000011 11000000 00000000

The result of right shifting
    960 = 00000000 00000000 00000011 11000000
8 bit positions using the right shift operator >> is
      3 = 00000000 00000000 00000000 00000011
```

- fig10_13.c (3 of 3)

534

The bitwise assignment operators

Bitwise assignment operators	
&=	Bitwise AND assignment operator.
|=	Bitwise inclusive OR assignment operator.
^=	Bitwise exclusive OR assignment operator.
<<=	Left-shift assignment operator.
>>=	Right-shift assignment operator.

535

Operator precedence and associativity. (Part 1 of 2.)

Operators	Associativity	Type
`()` `[]` `.` `->`	left to right	highest
`+` `-` `++` `--` `!` `&` `*` `~` `sizeof` (*type*)	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	shiting
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&`	left to right	bitwise AND

536

Operator precedence and associativity. (Part 2 of 2.)

Operators	Associativity	Type
^	left to right	bitwise OR
|	left to right	bitwise OR
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= &= |= ^= <<= >>= %=	right to left	assignment
,	left to right	comma

537

OBJECTIVES

- To create, initialize and use structures
- To pass structures to functions by value and by reference (Typedef)
- Example: Card shuffling and dealing
- To create, initialize and use unions
- Bitwise operators and bit fields
- To create, initialize and use enumeration

538

Bit Fields

- Bit field
  - Member of a structure whose size (in bits) has been specified
  - Enable better memory utilization
  - Must be defined as `int` or `unsigned`
  - Cannot access individual bits
- Defining bit fields
  - Follow `unsigned` or `int` member with a colon (`:`) and an integer constant representing the width of the field
  - Example:

```
struct BitCard {
   unsigned face : 4;
   unsigned suit : 2;
   unsigned color : 1;
};
```

539

Bit Fields

- Unnamed bit field
  - Field used as padding in the structure
  - Nothing may be stored in the bits

```
struct Example {
   unsigned a : 13;
   unsigned   : 3;
   unsigned b : 4;
}
```

  - Unnamed bit field with zero width aligns next bit field to a new storage unit boundary

540

```c
/* Fig. 10.16: fig10_16.c
   Representing cards with bit fields in a struct */

#include <stdio.h>

/* bitCard structure definition with bit fields */
struct bitCard {
   unsigned face : 4;  /* 4 bits; 0-15 */
   unsigned suit : 2;  /* 2 bits; 0-3 */
   unsigned color : 1; /* 1 bit; 0-1 */
}; /* end struct bitCard */

typedef struct bitCard Card; /* new type name for struct bitCard */

void fillDeck( Card * const wDeck );   /* prototype */
void deal( const Card * const wDeck ); /* prototype */

int main( void )
{
   Card deck[ 52 ]; /* create array of Cards */

   fillDeck( deck );
   deal( deck );

   return 0; /* indicates successful termination */

} /* end main */
```

Bit fields determine how much memory each member of a structure can take up

- fig10_16.c (1 of 2)

541

```c
/* initialize Cards */
void fillDeck( Card * const wDeck )
{
   int i; /* counter */

   /* loop through wDeck */
   for ( i = 0; i <= 51; i++ ) {
      wDeck[ i ].face = i % 13;
      wDeck[ i ].suit = i / 13;
      wDeck[ i ].color = i / 26;
   } /* end for */

} /* end function fillDeck */

/* output cards in two column format; cards 0-25 subscripted with
   k1 (column 1); cards 26-51 subscripted k2 (column 2) */
void deal( const Card * const wDeck )
{
   int k1; /* subscripts 0-25 */
   int k2; /* subscripts 26-51 */

   /* loop through wDeck */
   for ( k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ ) {
      printf( "Card:%3d  Suit:%2d  Color:%2d   ",
         wDeck[ k1 ].face, wDeck[ k1 ].suit, wDeck[ k1 ].color );
      printf( "Card:%3d  Suit:%2d  Color:%2d\n",
         wDeck[ k2 ].face, wDeck[ k2 ].suit, wDeck[ k2 ].color );
   } /* end for */

} /* end function deal */
```

- fig10_16.c (2 of 2)

542

```
Card:  0  Suit: 0  Color: 0   Card:  0  Suit: 2  Color: 1
Card:  1  Suit: 0  Color: 0   Card:  1  Suit: 2  Color: 1
Card:  2  Suit: 0  Color: 0   Card:  2  Suit: 2  Color: 1
Card:  3  Suit: 0  Color: 0   Card:  3  Suit: 2  Color: 1
Card:  4  Suit: 0  Color: 0   Card:  4  Suit: 2  Color: 1
Card:  5  Suit: 0  Color: 0   Card:  5  Suit: 2  Color: 1
Card:  6  Suit: 0  Color: 0   Card:  6  Suit: 2  Color: 1
Card:  7  Suit: 0  Color: 0   Card:  7  Suit: 2  Color: 1
Card:  8  Suit: 0  Color: 0   Card:  8  Suit: 2  Color: 1
Card:  9  Suit: 0  Color: 0   Card:  9  Suit: 2  Color: 1
Card: 10  Suit: 0  Color: 0   Card: 10  Suit: 2  Color: 1
Card: 11  Suit: 0  Color: 0   Card: 11  Suit: 2  Color: 1
Card: 12  Suit: 0  Color: 0   Card: 12  Suit: 2  Color: 1
Card:  0  Suit: 1  Color: 0   Card:  0  Suit: 3  Color: 1
Card:  1  Suit: 1  Color: 0   Card:  1  Suit: 3  Color: 1
Card:  2  Suit: 1  Color: 0   Card:  2  Suit: 3  Color: 1
Card:  3  Suit: 1  Color: 0   Card:  3  Suit: 3  Color: 1
Card:  4  Suit: 1  Color: 0   Card:  4  Suit: 3  Color: 1
Card:  5  Suit: 1  Color: 0   Card:  5  Suit: 3  Color: 1
Card:  6  Suit: 1  Color: 0   Card:  6  Suit: 3  Color: 1
Card:  7  Suit: 1  Color: 0   Card:  7  Suit: 3  Color: 1
Card:  8  Suit: 1  Color: 0   Card:  8  Suit: 3  Color: 1
Card:  9  Suit: 1  Color: 0   Card:  9  Suit: 3  Color: 1
Card: 10  Suit: 1  Color: 0   Card: 10  Suit: 3  Color: 1
Card: 11  Suit: 1  Color: 0   Card: 11  Suit: 3  Color: 1
Card: 12  Suit: 1  Color: 0   Card: 12  Suit: 3  Color: 1
```

543

OBJECTIVES

- To create, initialize and use structures
- To pass structures to functions by value and by reference (Typedef)
- Example: Card shuffling and dealing
- To create, initialize and use unions
- Bitwise operators and bit fields
- To create, initialize and use enumeration

544

Enumeration Constants

- Enumeration
  - Set of integer constants represented by identifiers
  - Enumeration constants are like symbolic constants whose values are automatically set
    - Values start at 0 and are incremented by 1
    - Values can be set explicitly with =
    - Need unique constant names
  - Example:

    ```
    enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG,
       SEP, OCT, NOV, DEC};
    ```

    - Creates a new type enum Months in which the identifiers are set to the integers 1 to 12
  - Enumeration variables can only assume their enumeration constant values (not the integer representations)

545

```
/* Fig. 10.18: fig10_18.c
   Using an enumeration type */
#include <stdio.h>

/* enumeration constants represent months of the year */
enum months {
   JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };

int main( void )
{
   enum months month; /* can contain any of the 12 months */

   /* initialize array of pointers */
   const char *monthName[] = { "", "January", "February", "March",
      "April", "May", "June", "July", "August", "September", "October",
      "November", "December" };

```

Enumeration sets the value of constant **JAN** to 1 and the following constants to 2, 3, 4…

- fig10_18.c (1 of 2)

546

```
   /* loop through months */
   for ( month = JAN; month <= DEC; month++ ) {
      printf( "%2d%11s\n", month, monthName[ month ] );
   } /* end for */

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

```
 1    January
 2   February
 3      March
 4      April
 5        May
 6       June
 7       July
 8     August
 9  September
10    October
11   November
12   December
```

Like symbolic constants, enumeration constants are replaced by their values at compile time

- fig10_18.c (2 of 2)

547

Review

- Structures are collections of related variables under one name. Structures may contain variables of many different data types in contrast to arrays that contain only elements of the same data type.
- Structures are derived data types (they are constructed using objects of other types).
- Members of the same structure type must have unique names, but two different structure types may contain members of the same name without conflict.
- Each structure definition must end with a semicolon.
- A structure cannot contain an instance of itself but may include a pointer to another object of the same type.

548

Review

- Structure definitions do not reserve any space in memory; rather, each definition creates a new data type that is used to define variables. Structure variables are defined like variables of other types.
- If there are fewer initializers in the list than members in the structure, the remaining members are automatically initialized to 0 (or NULL if the member is a pointer).
- Dot operator and arrow operator: highest operator precedence
- Structures may be passed to functions by passing individual structure members, by passing an entire structure or by passing a pointer to a structure.
- To pass structures by reference, pass the address of the structure variable.

549

Review

- The keyword "typedef" provides a mechanism for creating synonyms (alias) for previously defined data types.
- A union is a derived data type with members that share the storage space.
- On most systems, a sequence of 8 bits form a byte, the standard storage unit for a variable of type char. Other data types are stored in larger numbers of bytes.
- Bitwise operators and bit fields.
- A enumeration is a set of integer enumeration constants represented by identifiers.

550

The End

- Thank you very much!

551

Chapter 11

C File Processing

Acknowledgment

The notes are adapted from those provided by Deitel & Associates, Inc. and Pearson Education Inc.

552

OBJECTIVES

- To create, read, write and update files.
- Two file processing:
  - Sequential access file processing.
  - Random-access file processing.
- Review

553

Introduction

- Data files
  - Can be created, updated, and processed by C programs
  - Are used for permanent storage of large amounts of data
    - Storage of data in variables and arrays is only temporary

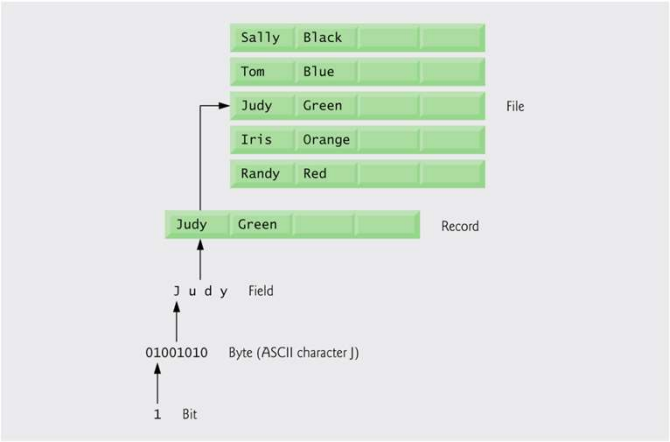
554

Data Hierarchy

- Data Hierarchy:
  - Bit – smallest data item
    - `Either value of 0` or `1`
  - Byte – 8 bits
    - Used to store a character
      - Decimal digits, letters, and special symbols
  - Field – group of characters conveying meaning
    - Example: your name
  - Record – group of related fields
    - Represented by a `struct`
    - Example: In a payroll system, a record for a particular employee includes his/her identification number, name, address, etc.
  - File – group of related records
    - Example: a payroll file
  - Database – the group of related files

555

Data hierarchy



556

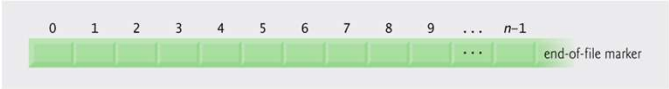
Data Hierarchy

- Data files
  - Record key
    - Identifies a record to facilitate the retrieval of specific records from a file
  - Sequential file
    - Records typically sorted by key

557

Files and Streams

- C views each file as a sequential stream of bytes
  - File ends with the *end-of-file marker*
    - Or, file ends at a specified byte



- When a file is opened, a stream is associated with the file:
  - Provide communication channel between files and programs
  - Opening a file returns a pointer to a `FILE` structure
    - Example file pointers:

      Pointers Files
    - `stdin` - standard input (keyboard)
    - `stdout` - standard output (screen)
    - `stderr` - standard error (screen)

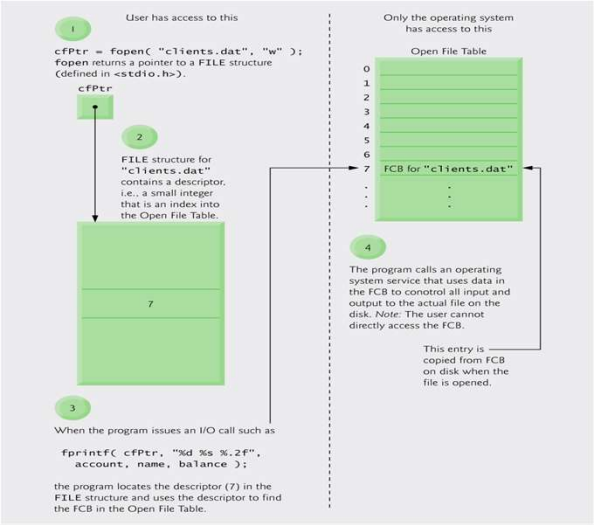
558

Files and Streams

- `FILE` structure
  - File descriptor
    - Index into operating system array called the open file table
  - File Control Block (FCB)
    - Found in every array element, system uses it to administer the file

559

Relationship between `FILE` pointers, `FILE` structures and FCBs



560

Files and Streams

- Read/Write functions in standard library
  - `fgetc`
    - Reads one character from a file
    - Takes a `FILE` pointer as an argument
    - `fgetc( stdin )` equivalent to `getchar()`
  - `fputc`
    - Writes one character to a file
    - Takes a `FILE` pointer and a character to write as an argument
    - `fputc( 'a', stdout )` equivalent to `putchar( 'a' )`
  - `fgets`
    - Reads a line from a file
  - `fputs`
    - Writes a line to a file
  - `fscanf` / `fprintf`
    - File processing equivalents of `scanf` and `printf`

561

OBJECTIVES

- To create, read, write and update files.
- Two file processing:
  - Sequential access file processing.
  - Random-access file processing.
- Review

562

Creating a Sequential-Access File

- C imposes no file structure
  - No notion of records in a file
  - Programmer must provide file structure
- Creating a File
  - `FILE *cfPtr;`
    - Creates a `FILE` pointer called `cfPtr`
  - `cfPtr = fopen("clients.dat", "w");`
    - Function `fopen` returns a `FILE` pointer to file specified
    - Takes two arguments – file to open and file open mode
    - If open fails, `NULL` returned

563

Creating a Sequential-Access File

- `fprintf`
  - **Used to print to a file**
  - **Like `printf`, except first argument is a `FILE` pointer (pointer to the file you want to print in)**
- `feof(` ***`FILE pointer`*** `)`
  - **Returns true if end-of-file indicator (no more data to process) is set for the specified file**
- `fclose(` ***`FILE pointer`*** `)`
  - **Closes specified file**
  - **Performed automatically when program ends**
  - **Good practice to close files explicitly**
- **Details**
  - **Programs may process no files, one file, or many files**
  - **Each file must have a unique name and should have its own pointer**

564

```c
/* Fig. 11.3: fig11_03.c
   Create a sequential file */
#include <stdio.h>

int main( void )
{
   int account;     /* account number */
   char name[ 30 ]; /* account name */
   double balance;  /* account balance */

   FILE *cfPtr;     /* cfPtr = clients.dat file pointer */

   /* fopen opens file. Exit program if unable to create file  */
   if ( ( cfPtr = fopen( "clients.dat", "w" ) ) == NULL ) {
      printf( "File could not be opened\n" );
   } /* end if */
   else {
      printf( "Enter the account, name, and balance.\n" );
      printf( "Enter EOF to end input.\n" );
      printf( "? " );
      scanf( "%d%s%lf", &account, name, &balance );

```

FILE pointer definition creates new file pointer

fopen function opens a file; **w** argument means the file is opened for writing

- fig11_03.c (1 of 2)


```c
      /* write account, name and balance into file with fprintf */
      while ( !feof( stdin ) ) {
         fprintf( cfPtr, "%d %s %.2f\n", account, name, balance );
         printf( "? " );
         scanf( "%d%s%lf", &account, name, &balance );
      } /* end while */

      fclose( cfPtr ); /* fclose closes file */
   } /* end else */

   return 0; /* indicates successful termination */

} /* end main */
```

```
Enter the account, name, and balance.
Enter EOF to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

feof returns true when end of file is reached

fprintf writes a string to a file

fclose closes a file

- fig11_03.c (2 of 2)

End-of-file key combinations for various popular operating systems.

Operating system	Key combination
Linux/Mac OS X/UNIX	*<Ctrl> d*
Windows	*<Ctrl> z*

567

File opening modes

Mode	Description
`r`	Open an existing file for reading.
`w`	Create a file for writing. If the file already exists, discard the current contents.
`a`	Append; open or create a file for writing at the end of the file.
`r+`	Open an existing file for update (reading and writing).
`w+`	Create a file for update. If the file already exists, discard the current contents.
`a+`	Append: open or create a file for update; writing is done at the end of the file.
`rb`	Open an existing file for reading in binary mode.
`wb`	Create a file for writing in binary mode. If the file already exists, discard the current contents.
`ab`	Append; open or create a file for writing at the end of the file in binary mode.
`rb+`	Open an existing file for update (reading and writing) in binary mode.
`wb+`	Create a file for update in binary mode. If the file already exists, discard the current contents.
`ab+`	Append: open or create a file for update in binary mode; writing is done at the end of the file.

568

Reading Data from a Sequential-Access File

- Reading a sequential access file
  - Create a `FILE` pointer, link it to the file to read
    `cfPtr = fopen( “clients.dat", "r" );`
  - Use `fscanf` to read from the file
    - Like `scanf`, except first argument is a `FILE` pointer
    `fscanf( cfPtr, "%d%s%f", &account, name, &balance );`
  - Data read from beginning to end
  - File position pointer
    - Indicates number of next byte to be read / written
    - Not really a pointer, but an integer value (specifies byte location)
    - Also called byte offset
  - `rewind( cfPtr )`
    - Repositions file position pointer to beginning of file (byte 0)

569

```c
/* Fig. 11.7: fig11_07.c
   Reading and printing a sequential file */
#include <stdio.h>

int main( void )
{
   int account;     /* account number */
   char name[ 30 ]; /* account name */
   double balance;  /* account balance */

   FILE *cfPtr;     /* cfPtr = clients.dat file pointer */

   /* fopen opens file; exits program if file cannot be opened */
   if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL ) {
      printf( "File could not be opened\n" );
   } /* end if */
   else { /* read account, name and balance from file */
      printf( "%-10s%-13s%s\n", "Account", "Name", "Balance" );
      fscanf( cfPtr, "%d%s%lf", &account, name, &balance );

```

fopen function opens a file; **r** argument means the file is opened for reading

- fig11_07.c
  (1 of 2)

570

```c
      /* while not end of file */
      while ( !feof( cfPtr ) ) {
         printf( "%-10d%-13s%7.2f\n", account, name, balance );
         fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
      } /* end while */

      fclose( cfPtr ); /* fclose closes the file */
   } /* end else */

   return 0; /* indicates successful termination */

} /* end main */
```

fscanf function reads a string from a file

```
Account   Name         Balance
100       Jones          24.98
200       Doe           345.67
300       White           0.00
400       Stone         -42.16
500       Rich          224.62
```

- fig11_07.c (2 of 2)

571

```c
/* Fig. 11.8: fig11_08.c
   Credit inquiry program */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int request;     /* request number */
   int account;     /* account number */
   double balance;  /* account balance */
   char name[ 30 ]; /* account name */
   FILE *cfPtr;     /* clients.dat file pointer */

   /* fopen opens the file; exits program if file cannot be opened */
   if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL ) {
      printf( "File could not be opened\n" );
   } /* end if */
   else {

      /* display request options */
      printf( "Enter request\n"
         " 1 - List accounts with zero balances\n"
         " 2 - List accounts with credit balances\n"
         " 3 - List accounts with debit balances\n"
         " 4 - End of run\n? " );
      scanf( "%d", &request );

```

- fig11_08.c (1 of 4)

572

```c
   /* process user's request */
   while ( request != 4 ) {

      /* read account, name and balance from file */
      fscanf( cfPtr, "%d%s%lf", &account, name, &balance );

      switch ( request ) {

         case 1:
            printf( "\nAccounts with zero balances:\n" );

            /* read file contents (until eof) */
            while ( !feof( cfPtr ) ) {

               if ( balance == 0 ) {
                  printf( "%-10d%-13s%7.2f\n",
                     account, name, balance );
               } /* end if */

               /* read account, name and balance from file */
               fscanf( cfPtr, "%d%s%lf",
                  &account, name, &balance );
            } /* end while */

            break;

```

- fig11_08.c (2 of 4)

573

```c
         case 2:
            printf( "\nAccounts with credit balances:\n" );

            /* read file contents (until eof) */
            while ( !feof( cfPtr ) ) {

               if ( balance < 0 ) {
                  printf( "%-10d%-13s%7.2f\n",
                     account, name, balance );
               } /* end if */

               /* read account, name and balance from file */
               fscanf( cfPtr, "%d%s%lf",
                  &account, name, &balance );
            } /* end while */

            break;

         case 3:
            printf( "\nAccounts with debit balances:\n" );

            /* read file contents (until eof) */
            while ( !feof( cfPtr ) ) {

               if ( balance > 0 ) {
                  printf( "%-10d%-13s%7.2f\n",
                     account, name, balance );
               } /* end if */

```

- fig11_08.c (3 of 4)

574

```c
                /* read account, name and balance from file */
                fscanf( cfPtr, "%d%s%lf",
                   &account, name, &balance );
             } /* end while */

             break;

          } /* end switch */

          rewind( cfPtr ); /* return cfPtr to beginning of file */

          printf( "\n? " );
          scanf( "%d", &request );
       } /* end while */

       printf( "End of run.\n" );
       fclose( cfPtr ); /* fclose closes the file */
    } /* end else */

    return 0; /* indicates successful termination */

} /* end main */
```

rewind function moves the file pointer back to the beginning of the file

- fig11_08.c (4 of 4)

575

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 1

Accounts with zero balances:
300       White           0.00

? 2

Accounts with credit balances:
400       Stone         -42.16

? 3

Accounts with debit balances:
100       Jones          24.98
200       Doe           345.67
500       Rich          224.62

? 4
End of run.
```

576

Reading Data from a Sequential-Access File

- Sequential access file
  - Cannot be modified without the risk of destroying other data
  - Fields can vary in size
    - Different representation in files and screen than internal representation
    - `1`, `34`, `-890` are all `int`s, but have different sizes on disk

Importance Notice!!

Sequential access with *fprintf* and *fscanf* is not usually used to **update** records.

577

OBJECTIVES

- To create, read, write and update files.
- Two file processing:
  - Sequential access file processing.
  - Random-access file processing.
- Review

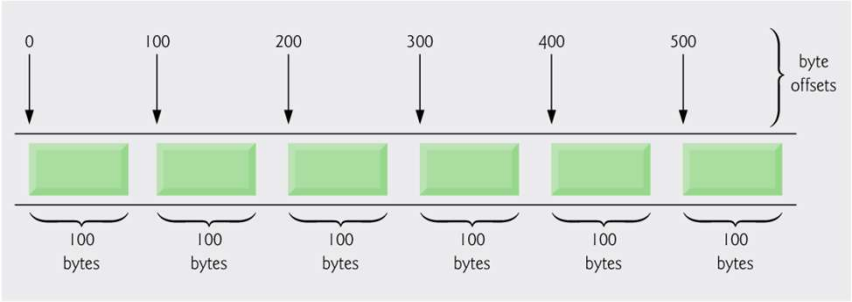
578

Random-Access Files

- Random access files
  - Access individual records without searching through other records
  - Instant access to records in a file
  - Data can be inserted without destroying other data
  - Data previously stored can be updated or deleted without overwriting
- Implemented using fixed length records
  - Sequential files do not have fixed length records

579

C's view of a random-access file



580

Creating a Random-Access File

- Data in random access files
  - Unformatted (stored as "raw bytes")
    - All data of the same type (`int`s, for example) uses the same amount of memory
    - All records of the same type have a fixed length. Hence, the exact location of a record relative to the beginning of the file can be calculated as a function of the record key.
    - Data not human readable

581

Creating a Random-Access File

- Unformatted I/O functions
  - `fwrite`
    - Transfer bytes from a location in memory to a file
  - `fread`
    - Transfer bytes from a file to a location in memory
  - Example:

    ```
    fwrite( &number, sizeof( int ), 1, myPtr );
    ```

    - `&number` – Location to transfer bytes from
    - `sizeof( int )` – Number of bytes to transfer
    - `1` – For arrays, number of elements to transfer
      - In this case, "one element" of an array is being transferred
    - `myPtr` – File to transfer to or from

582

Creating a Random-Access File

- Writing `structs`
  `fwrite( &myObject, sizeof (struct myStruct), 1, myPtr );`
  - `sizeof` – returns size in bytes of object in parentheses
- To write several array elements
  - Pointer to array as first argument
  - Number of elements to write as third argument

583

```c
/* Fig. 11.11: fig11_11.c
   Creating a random-access file sequentially */
#include <stdio.h>

/* clientData structure definition */
struct clientData {
   int acctNum;           /* account number */
   char lastName[ 15 ];  /* account last name */
   char firstName[ 10 ]; /* account first name */
   double balance;        /* account balance */
}; /* end structure clientData */

int main( void )
{
   int i; /* counter used to count from 1-100 */

   /* create clientData with default information */
   struct clientData blankClient = { 0, "", "", 0.0 };

```

- fig11_11.c
  (1 of 2)

584

```
   FILE *cfPtr; /* credit.dat file pointer */

   /* fopen opens the file; exits if file cannot be opened */
   if ( ( cfPtr = fopen( "credit.dat", "wb" ) ) == NULL ) {
      printf( "File could not be opened.\n" );
   } /* end if */
   else {

      /* output 100 blank records to file */
      for ( i = 1; i <= 100; i++ ) {
         fwrite( &blankClient, sizeof( struct clientData ), 1, cfPtr );
      } /* end for */

      fclose ( cfPtr ); /* fclose closes the file */
   } /* end else */

   return 0; /* indicates successful termination */

} /* end main */
```

fopen function opens a file; **wb** argument means the file is opened for writing in binary mode

fwrite transfers bytes into a random-access file

- fig11_11.c (2 of 2)

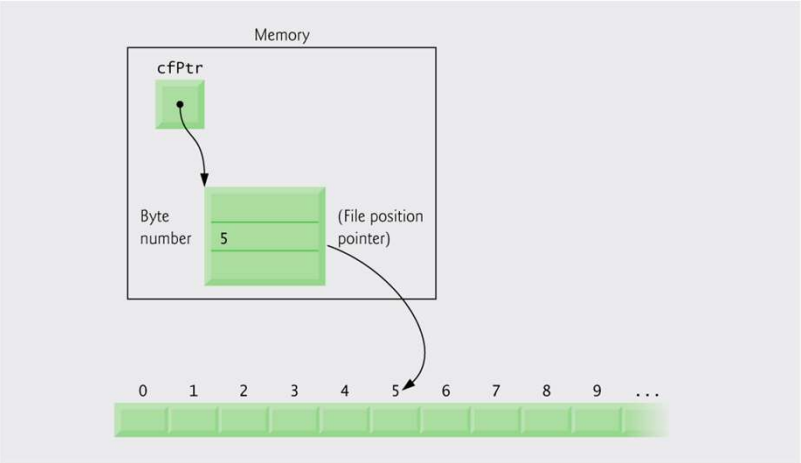
585

Writing Data Randomly to a Random-Access File

- `fseek`
  - Sets file position pointer to a specific position
  - `fseek(` *pointer, offset, symbolic_constant* `);`
    - *pointer* – pointer to file
    - *offset* – file position pointer (0 is first location)
    - *symbolic_constant* – specifies where in file we are reading from
      - `SEEK_SET` – seek starts at beginning of file
      - `SEEK_CUR` – seek starts at current location in file
      - `SEEK_END` – seek starts at end of file
  - Function fseek returns a nonzero value if the seek operation cannot be performed.

586

File position pointer indicating an offset of 5 bytes from the beginning of the file



587

```c
/* Fig. 11.12: fig11_12.c
   Writing to a random access file */
#include <stdio.h>

/* clientData structure definition */
struct clientData {
   int acctNum;           /* account number */
   char lastName[ 15 ];  /* account last name */
   char firstName[ 10 ]; /* account first name */
   double balance;        /* account balance */
}; /* end structure clientData */

int main( void )
{
   FILE *cfPtr; /* credit.dat file pointer */

   /* create clientData with default information */
   struct clientData client = { 0, "", "", 0.0 };

   /* fopen opens the file; exits if file cannot be opened */
   if ( ( cfPtr = fopen( "credit.dat", "rb+" ) ) == NULL ) {
      printf( "File could not be opened.\n" );
   } /* end if */
   else {

      /* require user to specify account number */
      printf( "Enter account number"
         " ( 1 to 100, 0 to end input )\n? " );
      scanf( "%d", &client.acctNum );

```

- fig11_12.c (1 of 2)

588

```c
      /* user enters information, which is copied into file */
      while ( client.acctNum != 0 ) {

         /* user enters last name, first name and balance */
         printf( "Enter lastname, firstname, balance\n? " );

         /* set record lastName, firstName and balance value */
         fscanf( stdin, "%s%s%lf", client.lastName,
            client.firstName, &client.balance );

         /* seek position in file to user-specified rec
         fseek( cfPtr, ( client.acctNum - 1 ) *
            sizeof( struct clientData ), SEEK_SET );

         /* write user-specified information in file */
         fwrite( &client, sizeof( struct clientData ), 1, cfPtr );

         /* enable user to input another account number */
         printf( "Enter account number\n? " );
         scanf( "%d", &client.acctNum );
      } /* end while */

      fclose( cfPtr ); /* fclose closes the file */
   } /* end else */

   return 0; /* indicates successful termination */

} /* end main */
```

fseek searches for a specific location in the random-access file

- fig11_12.c (2 of 2)

589

```
Enter account number ( 1 to 100, 0 to end input )
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0
```

590

Reading Data from a Random-Access File

- `fread`
  - Reads a specified number of bytes from a file into memory
    `fread( &client, sizeof (struct clientData), 1, myPtr );`
  - Can read several fixed-size array elements
    - Provide pointer to array
    - Indicate number of elements to read
  - To read multiple elements, specify in third argument

591

```c
/* Fig. 11.15: fig11_15.c
   Reading a random access file sequentially */
#include <stdio.h>

/* clientData structure definition */
struct clientData {
   int acctNum;          /* account number */
   char lastName[ 15 ];  /* account last name */
   char firstName[ 10 ]; /* account first name */
   double balance;       /* account balance */
}; /* end structure clientData */

int main( void )
{
   FILE *cfPtr; /* credit.dat file pointer */

   /* create clientData with default information */
   struct clientData client = { 0, "", "", 0.0 };

   /* fopen opens the file; exits if file cannot be opened */
   if ( ( cfPtr = fopen( "credit.dat", "rb" ) ) == NULL ) {
      printf( "File could not be opened.\n" );
   } /* end if */
```

- fig11_15.c (1 of 2)

592

```c
   else {
      printf( "%-6s%-16s%-11s%10s\n", "Acct", "Last Name",
         "First Name", "Balance" );

      /* read all records from file (until eof) */
      while ( !feof( cfPtr ) ) {
         fread( &client, sizeof( struct clientData ), 1, cfPtr );

         /* display record */
         if ( client.acctNum != 0 ) {
            printf( "%-6d%-16s%-11s%10.2f\n",
               client.acctNum, client.lastName,
               client.firstName, client.balance );
         } /* end if */

      } /* end while */

      fclose( cfPtr ); /* fclose closes the file */
   } /* end else */

   return 0; /* indicates successful termination */

} /* end main */
```

fread reads bytes from a random-access file to a location in memory

```
Acct  Last Name       First Name    Balance
29    Brown           Nancy          -24.54
33    Dunn            Stacey         314.33
37    Barker          Doug             0.00
88    Smith           Dave           258.34
96    Stone           Sam             34.98
```

- fig11_15.c (2 of 2)

593

```c
/* Fig. 11.16: fig11_16.c
   This program reads a random access file sequentially, updates data
   already written to the file, creates new data to be placed in the
   file, and deletes data previously in the file. */
#include <stdio.h>

/* clientData structure definition */
struct clientData {
   int acctNum;          /* account number */
   char lastName[ 15 ];  /* account last name */
   char firstName[ 10 ]; /* account first name */
   double balance;       /* account balance */
}; /* end structure clientData */

/* prototypes */
int enterChoice( void );
void textFile( FILE *readPtr );
void updateRecord( FILE *fPtr );
void newRecord( FILE *fPtr );
void deleteRecord( FILE *fPtr );

int main( void )
{
   FILE *cfPtr; /* credit.dat file pointer */
   int choice;  /* user's choice */

   /* fopen opens the file; exits if file cannot be opened */
   if ( ( cfPtr = fopen( "credit.dat", "rb+" ) ) == NULL ) {
      printf( "File could not be opened.\n" );
   } /* end if */
```

- fig11_16.c (1 of 10)

594

```c
   else {

      /* enable user to specify action */
      while ( ( choice = enterChoice() ) != 5 ) {

         switch ( choice ) {

            /* create text file from record file */
            case 1:
               textFile( cfPtr );
               break;

            /* update record */
            case 2:
               updateRecord( cfPtr );
               break;

            /* create record */
            case 3:
               newRecord( cfPtr );
               break;

            /* delete existing record */
            case 4:
               deleteRecord( cfPtr );
               break;

```

- fig11_16.c (2 of 10)

595

```c
            /* display message if user does not select valid choice */
            default:
               printf( "Incorrect choice\n" );
               break;

         } /* end switch */

      } /* end while */

      fclose( cfPtr ); /* fclose closes the file */
   } /* end else */

   return 0; /* indicates successful termination */

} /* end main */

/* create formatted text file for printing */
void textFile( FILE *readPtr )
{
   FILE *writePtr; /* accounts.txt file pointer */

   /* create clientData with default information */
   struct clientData client = { 0, "", "", 0.0 };

   /* fopen opens the file; exits if file cannot be opened */
   if ( ( writePtr = fopen( "accounts.txt", "w" ) ) == NULL ) {
      printf( "File could not be opened.\n" );
   } /* end if */
```

Function **textFile** creates a text file containing all account data

- fig11_16.c (3 of 10)

596

```c
   else {
      rewind( readPtr ); /* sets pointer to beginning of file */
      fprintf( writePtr, "%-6s%-16s%-11s%10s\n",
         "Acct", "Last Name", "First Name","Balance" );

      /* copy all records from random-access file into text file */
      while ( !feof( readPtr ) ) {
         fread( &client, sizeof( struct clientData ), 1, readPtr );

         /* write single record to text file */
         if ( client.acctNum != 0 ) {
            fprintf( writePtr, "%-6d%-16s%-11s%10.2f\n",
               client.acctNum, client.lastName,
               client.firstName, client.balance );
         } /* end if */

      } /* end while */

      fclose( writePtr ); /* fclose closes the file */
   } /* end else */

} /* end function textFile */

 /* update balance in record */
 void updateRecord( FILE *fPtr )
{
   int account;        /* account number */
   double transaction; /* transaction amount */

```

Function **updateRecord** changes the balance of a specified account

- fig11_16.c (4 of 10)

597

```c
   /* create clientData with no information */
   struct clientData client = { 0, "", "", 0.0 };

   /* obtain number of account to update */
   printf( "Enter account to update ( 1 - 100 ): " );
   scanf( "%d", &account );

   /* move file pointer to correct record in file */
   fseek( fPtr, ( account - 1 ) * sizeof( struct clientData ),
      SEEK_SET );

   /* read record from file */
   fread( &client, sizeof( struct clientData ), 1, fPtr );

   /* display error if account does not exist */
   if ( client.acctNum == 0 ) {
      printf( "Acount #%d has no information.\n", account );
   } /* end if */
   else { /* update record */
      printf( "%-6d%-16s%-11s%10.2f\n\n",
         client.acctNum, client.lastName,
         client.firstName, client.balance );

      /* request transaction amount from user */
      printf( "Enter charge ( + ) or payment ( - ): " );
      scanf( "%lf", &transaction );
      client.balance += transaction; /* update record balance */

```

- fig11_16.c (5 of 10)

598

```c
      printf( "%-6d%-16s%-11s%10.2f\n",
         client.acctNum, client.lastName,
         client.firstName, client.balance );

      /* move file pointer to correct record in file */
      fseek( fPtr, ( account - 1 ) * sizeof( struct clientData ),
         SEEK_SET );

      /* write updated record over old record in file */
      fwrite( &client, sizeof( struct clientData ), 1, fPtr );
   } /* end else */

} /* end function updateRecord */

 /* delete an existing record */
 void deleteRecord( FILE *fPtr )
{

   struct clientData client; /* stores record read from file */
   struct clientData blankClient = { 0, "", "", 0 }; /* blank client */

   int accountNum; /* account number */

   /* obtain number of account to delete */
   printf( "Enter account number to delete ( 1 - 100 ): " );
   scanf( "%d", &accountNum );

```

Function **deleteRecord** removes an existing account from the file

- fig11_16.c (6 of 10)

599

```c
   /* move file pointer to correct record in file */
   fseek( fPtr, ( accountNum - 1 ) * sizeof( struct clientData ),
      SEEK_SET );

   /* read record from file */
   fread( &client, sizeof( struct clientData ), 1, fPtr );

   /* display error if record does not exist */
   if ( client.acctNum == 0 ) {
      printf( "Account %d does not exist.\n", accountNum );
   } /* end if */
   else { /* delete record */

      /* move file pointer to correct record in file */
      fseek( fPtr, ( accountNum - 1 ) * sizeof( struct clientData ),
         SEEK_SET );

      /* replace existing record with blank record */
      fwrite( &blankClient,
         sizeof( struct clientData ), 1, fPtr );
   } /* end else */

 } /* end function deleteRecord */

```

- fig11_16.c (7 of 10)

600

```c
/* create and insert record */
void newRecord( FILE *fPtr )
{
   /* create clientData with default information */
   struct clientData client = { 0, "", "", 0.0 };

   int accountNum; /* account number */

   /* obtain number of account to create */
   printf( "Enter new account number ( 1 - 100 ): " );
   scanf( "%d", &accountNum );

   /* move file pointer to correct record in file */
   fseek( fPtr, ( accountNum - 1 ) * sizeof( struct clientData ),
      SEEK_SET );

   /* read record from file */
   fread( &client, sizeof( struct clientData ), 1, fPtr );

   /* display error if account already exists */
   if ( client.acctNum != 0 ) {
      printf( "Account #%d already contains information.\n",
         client.acctNum );
   } /* end if */
```

Function **newRecord** adds a new account to the file

- fig11_16.c (8 of 10)

601

```c
   else { /* create record */

      /* user enters last name, first name and balance */
      printf( "Enter lastname, firstname, balance\n? " );
      scanf( "%s%s%lf", &client.lastName, &client.firstName,
         &client.balance );

      client.acctNum = accountNum;

      /* move file pointer to correct record in file */
      fseek( fPtr, ( client.acctNum - 1 ) *
         sizeof( struct clientData ), SEEK_SET );

      /* insert record in file */
      fwrite( &client,
         sizeof( struct clientData ), 1, fPtr );
   } /* end else */

} /* end function newRecord */
```

- fig11_16.c (9 of 10)

602

```c
/* enable user to input menu choice */
int enterChoice( void )
{
  int menuChoice; /* variable to store user's choice */

  /* display available options */
  printf( "\nEnter your choice\n"
     "1 - store a formatted text file of acounts called\n"
     "    \"accounts.txt\" for printing\n"
     "2 - update an account\n"
     "3 - add a new account\n"
     "4 - delete an account\n"
     "5 - end program\n? " );

  scanf( "%d", &menuChoice ); /* receive choice from user */

  return menuChoice;

} /* end function enterChoice */
```

- fig11_16.c (10 of 10)

603

OBJECTIVES

- To create, read, write and update files.
- Two file processing:
  - Sequential access file processing.
  - Random-access file processing.
- Review

604

Review

- FILE *filePtr FILE is a structure data type
- filePtr = fopen("filename", "r"); if there is not a file, then return NULL pointer.
- Check if this file can be opened or not
  - if ((cfPtr = fopen("word.dat", "w")) == NULL) {}
    else { }
- The files will be closed when program execution terminates, but all files should be explicitly closed with fclose(filePtr);
- Three streams stdin, stdout and stderror are opened automatically by C when program execution begins.
- Fields -> records -> files -> database: in a file, each line is a record and each column is a field
- Sequential file vs random access file

605

The End

- Thank you very much!

606

Chapter 12

C Data Structures

Acknowledgment

The notes are adapted from those provided by Deitel & Associates, Inc. and Pearson Education Inc.

607

OBJECTIVES

- To allocate and free memory dynamically for data objects
  - malloc
  - free
- To create and manipulate
  - linked lists
  - stacks
  - queues
  - binary trees

608

Introduction

- Dynamic data structures
  - Data structures that grow and shrink during execution
- Use dynamic data structures to perform
  - Linked lists
    - Allow insertions and removals anywhere
  - Stacks
    - Allow insertions and removals only at top of stack
  - Queues
    - Allow insertions at the back and removals from the front
  - Binary trees
    - High-speed searching and sorting of data and efficient elimination of duplicate data items

609

Self-Referential Structures

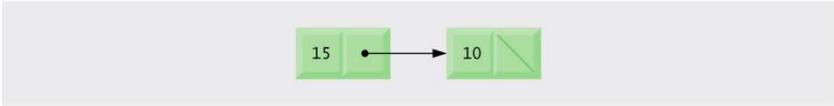
- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Can be linked together to form useful data structures such as lists, queues, stacks and trees
  - Terminated with a `NULL` pointer

```
struct node {
   int data;
   struct node *nextPtr;
}
```

- `nextPtr`
  - Points to an object of type `node`
  - Referred to as a link
    - Ties one `node` to another `node`

610

Self-referential structures linked together



611

OBJECTIVES

- To allocate and free memory dynamically for data objects
  - malloc
  - free
- To create and manipulate
  - linked lists
  - stacks
  - queues
  - binary trees

612

Dynamic Memory Allocation

- Dynamic memory allocation
  - Obtain and release memory during execution
- `malloc`
  - Takes number of bytes to allocate
    - Use `sizeof` to determine the size of an object
  - Returns pointer of type `void *`
    - A `void *` pointer may be assigned to any pointer
    - If no memory available, returns `NULL`
  - Example

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

- `free`
  - Deallocates memory allocated by `malloc`
  - Takes a pointer as an argument
  - `free ( newPtr );`
  - Notes:
    - `Freeing memory not allocated dynamically with malloc is an error;`
    - `Referring to memory that has been freed is an error.`

613

OBJECTIVES

- To allocate and free memory dynamically for data objects
  - malloc
  - free
- To create and manipulate
  - linked lists
  - stacks
  - queues
  - binary trees

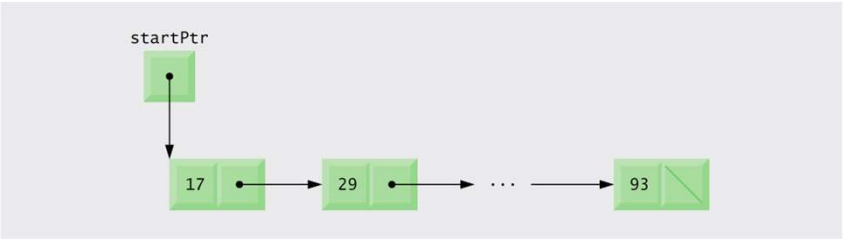
614

Linked Lists

- Linked list
  - Linear collection of self-referential class objects, called nodes
  - Connected by pointer links
  - Accessed via a pointer to the first node of the list
  - Subsequent nodes are accessed via the link-pointer member of the current node
  - Link pointer in the last node is set to `NULL` to mark the list's end
- Use a linked list instead of an array when
  - You have an unpredictable number of data elements
  - Your list needs to be sorted quickly

615

Linked list graphical representation



616

```c
/* Fig. 12.3: fig12_03.c
   Operating and maintaining a list */
#include <stdio.h>
#include <stdlib.h>

/* self-referential structure */
struct listNode {
   char data; /* each listNode contains a character */
   struct listNode *nextPtr; /* pointer to next node */
}; /* end structure listNode */

typedef struct listNode ListNode; /* synonym for struct listNode */
typedef ListNode *ListNodePtr; /* synonym for ListNode* */

/* prototypes */
void insert( ListNodePtr *sPtr, char value );
char delete( ListNodePtr *sPtr, char value );
int isEmpty( ListNodePtr sPtr );
void printList( ListNodePtr currentPtr );
void instructions( void );

int main( void )
{
   ListNodePtr startPtr = NULL; /* initially there are no nodes */
   int choice; /* user's choice */
   char item;  /* char entered by user */

   instructions(); /* display the menu */
   printf( "? " );
   scanf( "%d", &choice );
```

Each node in the list contains a data element and a pointer to the next node

- fig12_03.c (1 of 8)

617

```c

   /* loop while user does not choose 3 */
   while ( choice != 3 ) {

      switch ( choice ) {

         case 1:
            printf( "Enter a character: " );
            scanf( "\n%c", &item );
            insert( &startPtr, item ); /* insert item in list */
            printList( startPtr );
            break;

         case 2: /* delete an element */

            /* if list is not empty */
            if ( !isEmpty( startPtr ) ) {
               printf( "Enter character to be deleted: " );
               scanf( "\n%c", &item );

               /* if character is found, remove it*/
               if ( delete( &startPtr, item ) ) { /* remove item */
                  printf( "%c deleted.\n", item );
                  printList( startPtr );
               } /* end if */
               else {
                  printf( "%c not found.\n\n", item );
               } /* end else */

            } /* end if */
```

Function **insert** inserts data into the list

Function **delete** removes data from the list

- fig12_03.c (2 of 8)

618

```c
            else {
               printf( "List is empty.\n\n" );
            } /* end else */

            break;

         default:
            printf( "Invalid choice.\n\n" );
            instructions();
            break;

      } /* end switch */

      printf( "? " );
      scanf( "%d", &choice );
   } /* end while */

   printf( "End of run.\n" );

   return 0; /* indicates successful termination */

} /* end main */

```

- fig12_03.c (3 of 8)

619

```c
/* display program instructions to user */
void instructions( void )
{
   printf( "Enter your choice:\n"
      "   1 to insert an element into the list.\n"
      "   2 to delete an element from the list.\n"
      "   3 to end.\n" );
} /* end function instructions */

/* Insert a new value into the list in sorted order */
void insert( ListNodePtr *sPtr, char value )
{
   ListNodePtr newPtr;      /* pointer to new node */
   ListNodePtr previousPtr; /* pointer to previous node in list */
   ListNodePtr currentPtr;  /* pointer to current node in list */

   newPtr = malloc( sizeof( ListNode ) ); /* create node */

   if ( newPtr != NULL ) { /* is space available */
      newPtr->data = value; /* place value in node */
      newPtr->nextPtr = NULL; /* node does not link to another node */

      previousPtr = NULL;
      currentPtr = *sPtr;

      /* loop to find the correct location in the list */
      while ( currentPtr != NULL && value > currentPtr->data ) {
         previousPtr = currentPtr;          /* walk to ...   */
         currentPtr = currentPtr->nextPtr;  /* ... next node */
      } /* end while */
```

To insert a node into the list, memory must first be allocated for that node

while loop searches for new node's place in the list

- fig12_03.c **(4 of 8)**

620

```
   /* insert new node at beginning of list */
   if ( previousPtr == NULL ) {
      newPtr->nextPtr = *sPtr;
      *sPtr = newPtr;
   } /* end if */
   else { /* insert new node between previousPtr and currentPtr */
      previousPtr->nextPtr = newPtr;
      newPtr->nextPtr = currentPtr;
   } /* end else */

 } /* end if */
 else {
   printf( "%c not inserted. No memory available.\n", value );
 } /* end else */

} /* end function insert */

 /* Delete a list element */
char delete( ListNodePtr *sPtr, char value )
{
  ListNodePtr previousPtr; /* pointer to previous node in list */
  ListNodePtr currentPtr;  /* pointer to current node in list */
  ListNodePtr tempPtr;     /* temporary node pointer */

```

If there are no nodes in the list, the new node becomes the "start" node

Otherwise, the new node is inserted between two others (or at the end of the list) by changing pointers

- fig12_03.c **(5 of 8)**

621

```
  /* delete first node */
  if ( value == ( *sPtr )->data ) {
     tempPtr = *sPtr; /* hold onto node being removed */
     *sPtr = ( *sPtr )->nextPtr; /* de-thread the node */
     free( tempPtr ); /* free the de-threaded node */
     return value;
  } /* end if */
  else {
     previousPtr = *sPtr;
     currentPtr = ( *sPtr )->nextPtr;

     /* loop to find the correct location in the list */
     while ( currentPtr != NULL && currentPtr->data != value ) {
        previousPtr = currentPtr;         /* walk to ...   */
        currentPtr = currentPtr->nextPtr; /* ... next node */
     } /* end while */

     /* delete node at currentPtr */
     if ( currentPtr != NULL ) {
        tempPtr = currentPtr;
        previousPtr->nextPtr = currentPtr->nextPtr;
        free( tempPtr );
        return value;
     } /* end if */
```

`while` loop searches for node's place in the list

Once the node is found, it is deleted by changing pointers and freeing the node's memory

- fig12_03.c **(6 of 8)**

622

```
  } /* end else */

  return '\0';

} /* end function delete */

/* Return 1 if the list is empty, 0 otherwise */
int isEmpty( ListNodePtr sPtr )
{
  return sPtr == NULL;

} /* end function isEmpty */

/* Print the list */
```

If the start node is **NULL**, there are no nodes in the list

- fig12_03.c
  (7 of 8)

623

```
void printList( ListNodePtr currentPtr )
{

  /* if list is empty */
  if ( currentPtr == NULL ) {
     printf( "List is empty.\n\n" );
  } /* end if */
  else {
     printf( "The list is:\n" );

     /* while not the end of the list */
     while ( currentPtr != NULL ) {
        printf( "%c --> ", currentPtr->data );
        currentPtr = currentPtr->nextPtr;
     } /* end while */

     printf( "NULL\n\n" );
  } /* end else */

} /* end function printList */
```

- fig12_03.c
  (8 of 8)

624

```
Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 1
Enter a character: B
The list is:
B --> NULL

? 1
Enter a character: A
The list is:
A --> B --> NULL

? 1
Enter a character: C
The list is:
A --> B --> C --> NULL

? 2
Enter character to be deleted: D
D not found.

? 2
Enter character to be deleted: B
B deleted.
The list is:
A --> C --> NULL


? 2
Enter character to be deleted: C
C deleted.
The list is:
A --> NULL
```

(continued on next slide…)

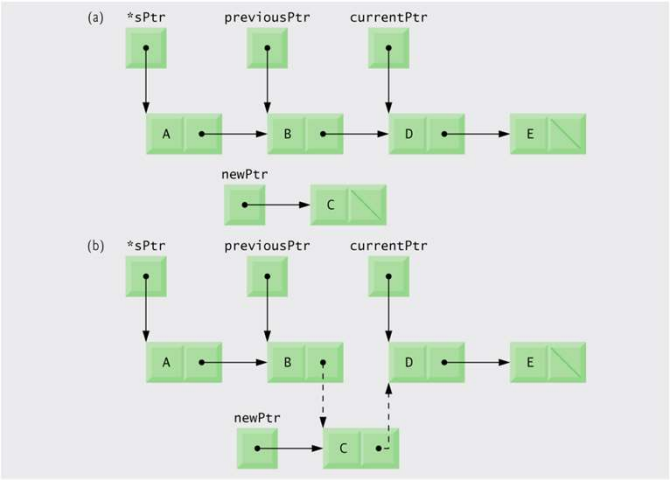
625

(continued from previous slide…)

```
? 2
Enter character to be deleted: A
A deleted.
List is empty.

? 4
Invalid choice.

Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 3
End of run.

? 2
Enter character to be deleted: C
C deleted.
The list is:
A --> NULL

? 2
Enter character to be deleted: A
A deleted.
List is empty.

? 4
Invalid choice.

Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 3
End of run.
```

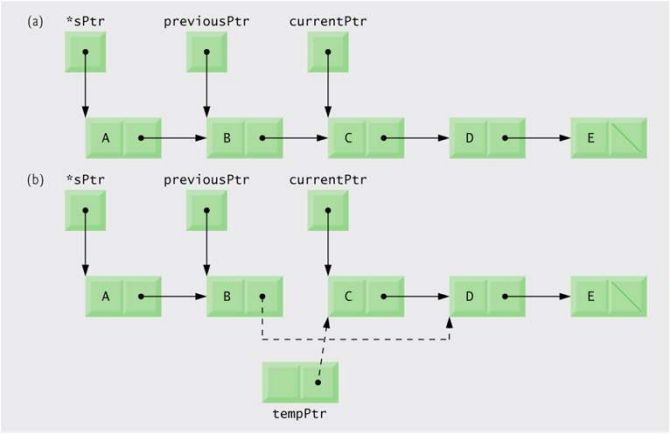
626

Inserting a node in order in a list



627

Deleting a node from a list



628

OBJECTIVES

- To allocate and free memory dynamically for data objects
  - malloc
  - free
- To create and manipulate
  - linked lists
  - stacks
  - queues
  - binary trees

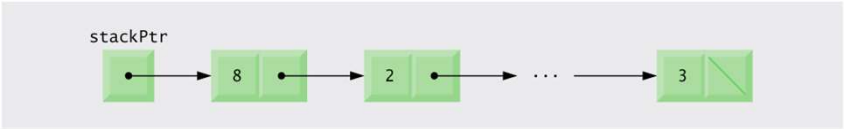
629

Stacks

- Stack
  - New nodes can be added and removed only at the top
  - Similar to a pile of dishes
  - Last-in, first-out (LIFO)
  - Bottom of stack indicated by a link member to `NULL`
  - Constrained version of a linked list
- `push`
  - Adds a new node to the top of the stack
- `pop`
  - Removes a node from the top
  - Stores the popped value
  - Returns `true` if `pop` was successful

630

Stack graphical representation



631

```c
/* Fig. 12.8: fig12_08.c
   dynamic stack program */
#include <stdio.h>
#include <stdlib.h>

/* self-referential structure */
struct stackNode {
   int data;              /* define data as an int */
   struct stackNode *nextPtr; /* stackNode pointer */
}; /* end structure stackNode */

typedef struct stackNode StackNode; /* synonym for struct stackNode */
typedef StackNode *StackNodePtr; /* synonym for StackNode* */

/* prototypes */
void push( StackNodePtr *topPtr, int info );
int pop( StackNodePtr *topPtr );
int isEmpty( StackNodePtr topPtr );
void printStack( StackNodePtr currentPtr );
void instructions( void );

/* function main begins program execution */
int main( void )
{
   StackNodePtr stackPtr = NULL; /* points to stack top */
   int choice; /* user's menu choice */
   int value;  /* int input by user */

   instructions(); /* display the menu */
   printf( "? " );
```

Each node in the stack contains a data element and a pointer to the next node

- fig12_08.c **(1 of 5)**

632

```c
   scanf( "%d", &choice );

   /* while user does not enter 3 */
   while ( choice != 3 ) {

      switch ( choice ) {

         /* push value onto stack */
         case 1:
            printf( "Enter an integer: " );
            scanf( "%d", &value );
            push( &stackPtr, value );
            printStack( stackPtr );
            break;

         /* pop value off stack */
         case 2:

            /* if stack is not empty */
            if ( !isEmpty( stackPtr ) ) {
               printf( "The popped value is %d.\n", pop( &stackPtr ) );
            } /* end if */

            printStack( stackPtr );
            break;

         default:
            printf( "Invalid choice.\n\n" );
            instructions();
            break;
```

- fig12_08.c **(2 of 5)**

633

```c

      } /* end switch */

      printf( "? " );
      scanf( "%d", &choice );
   } /* end while */

   printf( "End of run.\n" );

   return 0; /* indicates successful termination */

} /* end main */

/* display program instructions to user */
void instructions( void )
{
   printf( "Enter choice:\n"
      "1 to push a value on the stack\n"
      "2 to pop a value off the stack\n"
      "3 to end program\n" );
} /* end function instructions */

/* Insert a node at the stack top */
void push( StackNodePtr *topPtr, int info )
{
   StackNodePtr newPtr; /* pointer to new node */

   newPtr = malloc( sizeof( StackNode ) );

```

To insert a node into the stack, memory must first be allocated for that node

- fig12_08.c **(3 of 5)**

634

```c
   /* insert the node at stack top */
   if ( newPtr != NULL ) {
      newPtr->data = info;
      newPtr->nextPtr = *topPtr;
      *topPtr = newPtr;
   } /* end if */
   else { /* no space available */
      printf( "%d not inserted. No memory available.\n", info );
   } /* end else */

} /* end function push */

/* Remove a node from the stack top */
int pop( StackNodePtr *topPtr )
{
   StackNodePtr tempPtr; /* temporary node pointer */
   int popValue; /* node value */

   tempPtr = *topPtr;
   popValue = ( *topPtr )->data;
   *topPtr = ( *topPtr )->nextPtr;
   free( tempPtr );

   return popValue;

} /* end function pop */
```

Stack nodes are always inserted at the top, so there is no need to search for the node's place

Inserted node becomes the new top

Stack nodes are always removed from the top, so there is no need to search for the node's place

Second node becomes the new top

Free the memory of the popped node

- fig12_08.c **(4 of 5)**

635

```c
 /* Print the stack */
 void printStack( StackNodePtr currentPtr )
{

   /* if stack is empty */
   if ( currentPtr == NULL ) {
      printf( "The stack is empty.\n\n" );
   } /* end if */
   else {
      printf( "The stack is:\n" );

      /* while not the end of the stack */
      while ( currentPtr != NULL ) {
         printf( "%d --> ", currentPtr->data );
         currentPtr = currentPtr->nextPtr;
      } /* end while */

      printf( "NULL\n\n" );
   } /* end else */

} /* end function printList */
} /* end function printList */

/* Return 1 if the stack is empty, 0 otherwise */
int isEmpty( StackNodePtr topPtr )
{
   return topPtr == NULL;

} /* end function isEmpty */
```

- fig12_08.c **(5 of 5)**

636

```
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 1
Enter an integer: 5
The stack is:
5 --> NULL


Enter an integer: 6
The stack is:
6 --> 5 --> NULL

? 1
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL

? 2
The popped value is 4.
The stack is:
6 --> 5 --> NULL
```

(continued on next slide…)

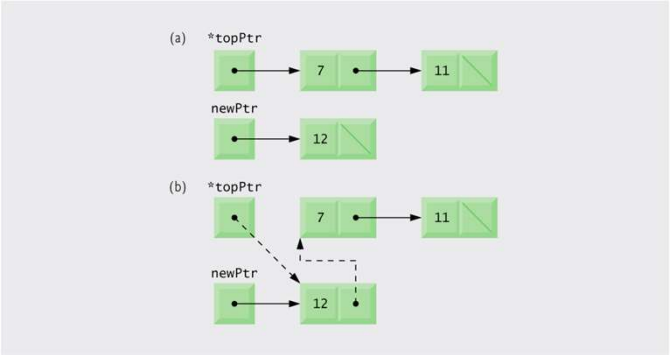
637

(continued from previous slide…)

```
? 2
The popped value is 6.
The stack is:
5 --> NULL

? 2
The popped value is 5.
The stack is empty.

? 2
The stack is empty.

? 4
Invalid choice.

Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 3
End of run.
```

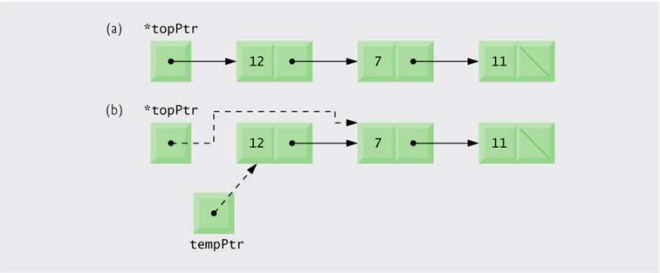
638

`push` operation



639

`pop` operation



640

OBJECTIVES

- To allocate and free memory dynamically for data objects
  - malloc
  - free
- To create and manipulate
  - linked lists
  - stacks
  - queues
  - binary trees

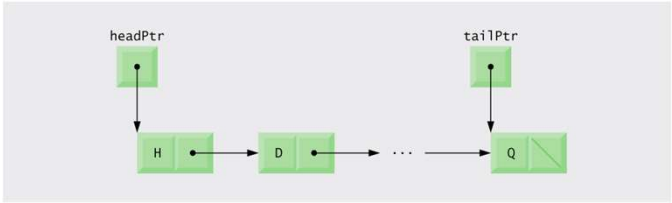
641

Queues

- Queue
  - Similar to a supermarket checkout line
  - First-in, first-out (FIFO)
  - Nodes are removed only from the head
  - Nodes are inserted only at the tail
- Insert and remove operations
  - enqueue (insert) and dequeue (remove)

642

Queue graphical representation



643

```c
/* Fig. 12.13: fig12_13.c
   Operating and maintaining a queue */

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

/* self-referential structure */
struct queueNode {
   char data;                  /* define data as a char */
   struct queueNode *nextPtr; /* queueNode pointer */
}; /* end structure queueNode */

typedef struct queueNode QueueNode;
typedef QueueNode *QueueNodePtr;

/* function prototypes */
void printQueue( QueueNodePtr currentPtr );
int isEmpty( QueueNodePtr headPtr );
char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr );
void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
              char value );
void instructions( void );

/* function main begins program execution */
int main( void )
{
   QueueNodePtr headPtr = NULL; /* initialize headPtr */
   QueueNodePtr tailPtr = NULL; /* initialize tailPtr */
   int choice;                  /* user's menu choice */
   char item;                   /* char input by user */
```

Each node in the queue contains a data element and a pointer to the next node

Note that unlike linked lists and stacks, queues keep track of the tail node as well as the head

- fig12_13.c **(1 of 6)**

644

```c

   instructions(); /* display the menu */
   printf( "? " );
   scanf( "%d", &choice );

   /* while user does not enter 3 */
   while ( choice != 3 ) {

      switch( choice ) {

         /* enqueue value */
         case 1:
            printf( "Enter a character: " );
            scanf( "\n%c", &item );
            enqueue( &headPtr, &tailPtr, item );
            printQueue( headPtr );
            break;

         /* dequeue value */
         case 2:

            /* if queue is not empty */
            if ( !isEmpty( headPtr ) ) {
               item = dequeue( &headPtr, &tailPtr );
               printf( "%c has been dequeued.\n", item );
            } /* end if */

            printQueue( headPtr );
            break;
```

- fig12_13.c **(2 of 6)**

645

```c

         default:
            printf( "Invalid choice.\n\n" );
            instructions();
            break;

      } /* end switch */

      printf( "? " );
      scanf( "%d", &choice );
   } /* end while */

   printf( "End of run.\n" );

   return 0; /* indicates successful termination */

} /* end main */

/* display program instructions to user */
void instructions( void )
{
   printf ( "Enter your choice:\n"
            "   1 to add an item to the queue\n"
            "   2 to remove an item from the queue\n"
            "   3 to end\n" );
} /* end function instructions */
```

- fig12_13.c **(3 of 6)**

646

```c

/* insert a node a queue tail */
void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
              char value )
{
   QueueNodePtr newPtr; /* pointer to new node */

   newPtr = malloc( sizeof( QueueNode ) );

   if ( newPtr != NULL ) { /* is space available */
      newPtr->data = value;
      newPtr->nextPtr = NULL;

      /* if empty, insert node at head */
      if ( isEmpty( *headPtr ) ) {
         *headPtr = newPtr;
      } /* end if */
      else {
         ( *tailPtr )->nextPtr = newPtr;
      } /* end else */

      *tailPtr = newPtr;
   } /* end if */
   else {
      printf( "%c not inserted. No memory available.\n", value );
   } /* end else */

} /* end function enqueue */
```

To insert a node into the queue, memory must first be allocated for that node

Queue nodes are always inserted at the tail, so there is no need to search for the node's place

If the queue is empty, the inserted node becomes the new head in addition to the new tail

Inserted node becomes the new tail

- fig12_13.c **(4 of 6)**

647

```c

/* remove node from queue head */
char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr )
 {
  char value;           /* node value */
  QueueNodePtr tempPtr; /* temporary node pointer */

  value = ( *headPtr )->data;
  tempPtr = *headPtr;
  *headPtr = ( *headPtr )->nextPtr;

  /* if queue is empty */
  if ( *headPtr == NULL ) {
     *tailPtr = NULL;
  } /* end if */

  free( tempPtr );

  return value;

} /* end function dequeue */

/* Return 1 if the list is empty, 0 otherwise */
int isEmpty( QueueNodePtr headPtr )
{
  return headPtr == NULL;

} /* end function isEmpty */
```

Queue nodes are always removed from the head, so there is no need to search for the node's place

Second node becomes the new head

If the removed node is the only node in the queue, it is the tail as well as the head of the queue, so **`tailPtr`** must be set to **`NULL`**

Free the memory of the removed node

- fig12_13.c **(5 of 6)**

648

```c

/* Print the queue */
void printQueue( QueueNodePtr currentPtr )
{

  /* if queue is empty */
  if ( currentPtr == NULL ) {
     printf( "Queue is empty.\n\n" );
  } /* end if */
  else {
     printf( "The queue is:\n" );

   /* while not end of queue */
     while ( currentPtr != NULL ) {
        printf( "%c --> ", currentPtr->data );
        currentPtr = currentPtr->nextPtr;
     } /* end while */

     printf( "NULL\n\n" );
  } /* end else */

} /* end function printQueue */
```

- fig12_13.c
  (6 of 6)

649

```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 1
Enter a character: A
The queue is:
A --> NULL

? 1
Enter a character: B
The queue is:
A --> B --> NULL

? 1
Enter a character: C
The queue is:
A --> B --> C --> NULL

? 2
A has been dequeued.
The queue is:
B --> C --> NULL
```

(continued on next slide…)

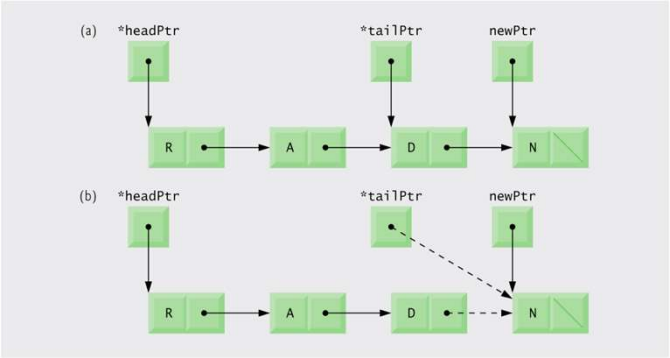
650

(continued from previous slide…)

```
? 2
B has been dequeued.
The queue is:
C --> NULL

? 2
C has been dequeued.
Queue is empty.

? 2
Queue is empty.

? 4
Invalid choice.

Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 3
End of run.
```

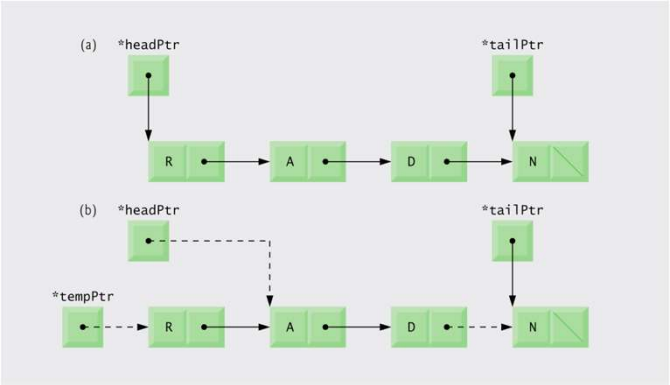
651

enqueue operation



652

dequeue operation



653

OBJECTIVES

- To allocate and free memory dynamically for data objects
  - malloc
  - free
- To create and manipulate
  - linked lists
  - stacks
  - queues
  - binary trees

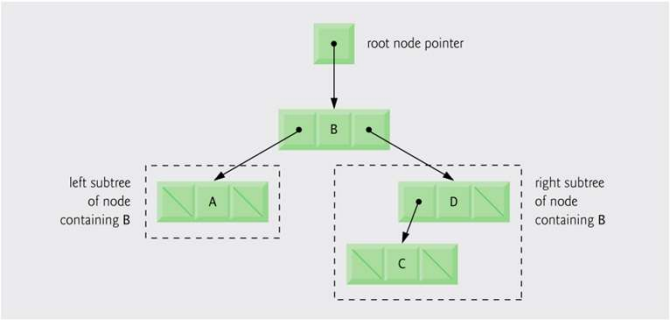
654

Trees

- Tree nodes contain two or more links
  - All other data structures we have discussed only contain one
- Binary trees
  - All nodes contain two links
    - None, one, or both of which may be NULL
  - The root node is the first node in a tree.
  - Each link in the root node refers to a child
  - A node with no children is called a leaf node

655

Binary tree graphical representation



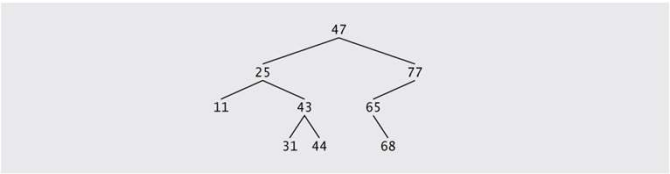
656

Trees

- Binary search tree
  - Values in left subtree less than parent
  - Values in right subtree greater than parent
  - Facilitates duplicate elimination
  - Fast searches - for a balanced tree, maximum of log n comparisons

657

Binary search tree



658

Trees

- Tree traversals:
  - Inorder traversal – prints the node values in ascending order
    1. Traverse the left subtree with an inorder traversal
    2. Process the value in the node (i.e., print the node value)
    3. Traverse the right subtree with an inorder traversal
  - Preorder traversal
    1. Process the value in the node
    2. Traverse the left subtree with a preorder traversal
    3. Traverse the right subtree with a preorder traversal
  - Postorder traversal
    1. Traverse the left subtree with a postorder traversal
    2. Traverse the right subtree with a postorder traversal
    3. Process the value in the node

659

```c
/* Fig. 12.19: fig12_19.c
   Create a binary tree and traverse it
   preorder, inorder, and postorder */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* self-referential structure */
struct treeNode {
   struct treeNode *leftPtr;  /* pointer to left subtree */
   int data; /* node value */
   struct treeNode *rightPtr; /* pointer to right subtree */
}; /* end structure treeNode */

typedef struct treeNode TreeNode; /* synonym for struct treeNode */
typedef TreeNode *TreeNodePtr; /* synonym for TreeNode* */

/* prototypes */
void insertNode( TreeNodePtr *treePtr, int value );
void inOrder( TreeNodePtr treePtr );
void preOrder( TreeNodePtr treePtr );
void postOrder( TreeNodePtr treePtr );

/* function main begins program execution */
int main( void )
{
   int i; /* counter to loop from 1-10 */
   int item; /* variable to hold random values */
   TreeNodePtr rootPtr = NULL; /* tree initially empty */

```

Each node in the tree contains a data element and a pointer to the left and right child nodes

- fig12_19.c **(1 of 5)**

660

```c
   srand( time( NULL ) );
   printf( "The numbers being placed in the tree are:\n" );

   /* insert random values between 0 and 14 in the tree */
   for ( i = 1; i <= 10; i++ ) {
      item = rand() % 15;
      printf( "%3d", item );
      insertNode( &rootPtr, item );
   } /* end for */

   /* traverse the tree preOrder */
   printf( "\n\nThe preOrder traversal is:\n" );
   preOrder( rootPtr );

   /* traverse the tree inOrder */
   printf( "\n\nThe inOrder traversal is:\n" );
   inOrder( rootPtr );

   /* traverse the tree postOrder */
   printf( "\n\nThe postOrder traversal is:\n" );
   postOrder( rootPtr );

   return 0; /* indicates successful termination */

} /* end main */
```

- fig12_19.c
  (2 of 5)

661

```c

/* insert node into tree */
void insertNode( TreeNodePtr *treePtr, int value )
{

   /* if tree is empty */
   if ( *treePtr == NULL ) {
      *treePtr = malloc( sizeof( TreeNode ) );

      /* if memory was allocated then assign data */
      if ( *treePtr != NULL ) {
         ( *treePtr )->data = value;
         ( *treePtr )->leftPtr = NULL;
         ( *treePtr )->rightPtr = NULL;
      } /* end if */
      else {
         printf( "%d not inserted. No memory available.\n", value );
      } /* end else */

   } /* end if */
   else { /* tree is not empty */

      /* data to insert is less than data in current node */
      if ( value < ( *treePtr )->data ) {
         insertNode( &( ( *treePtr )->leftPtr ), value );
      } /* end if */
```

To insert a node into the tree, memory must first be allocated for that node

If the inserted node's data is less than the current node's, the program will attempt to insert the node at the current node's left child.

- fig12_19.c
  (3 of 5)

662

```
82
83       /* data to insert is greater than data in current node */
84       else if ( value > ( *treePtr )->data ) {
85          insertNode( &( ( *treePtr )->rightPtr ), value );
86       } /* end else if */
87       else { /* duplicate data value ignored */
88          printf( "dup" );
89       } /* end else */
90
91    } /* end else */
92
93 } /* end function insertNode */
94
95 /* begin inorder traversal of tree */
96 void inOrder( TreeNodePtr treePtr )
97 {
98
99    /* if tree is not empty then traverse */
100   if ( treePtr != NULL ) {
101      inOrder( treePtr->leftPtr );
102      printf( "%3d", treePtr->data );
103      inOrder( treePtr->rightPtr );
104   } /* end if */
105
106 } /* end function inOrder */
107
108 /* begin preorder traversal of tree */
109 void preOrder( TreeNodePtr treePtr )
110 {
111
```

If the inserted node's data is greater than the current node's, the program will attempt to insert the node at the current node's right child.

The inorder traversal calls an inorder traversal on the node's left child, then prints the node itself, then calls an inorder traversal on the right child.

- fig12_19.c **(4 of 5)**

663

```
112   /* if tree is not empty then traverse */
113   if ( treePtr != NULL ) {
114      printf( "%3d", treePtr->data );
115      preOrder( treePtr->leftPtr );
116      preOrder( treePtr->rightPtr );
117   } /* end if */
118
119 } /* end function preOrder */
120
121 /* begin postorder traversal of tree */
122 void postOrder( TreeNodePtr treePtr )
123 {
124
125   /* if tree is not empty then traverse */
126   if ( treePtr != NULL ) {
127      postOrder( treePtr->leftPtr );
128      postOrder( treePtr->rightPtr );
129      printf( "%3d", treePtr->data );
130   } /* end if */
131
132 } /* end function postOrder */
```

The preorder traversal prints the node itself, then calls a preorder traversal on the node's left child, then calls a preorder traversal on the right child.

The postorder traversal calls an postorder traversal on the node's left child, then calls an postorder traversal on the right child, then prints the node itself.

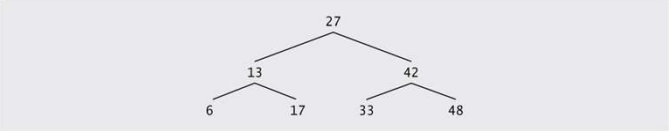
- fig12_19.c **(5 of 5)**

664

```
The numbers being placed in the tree are:
  6  7  4 12  7dup  2  2dup  5  7dup 11

The preOrder traversal is:
  6  4  2  5  7 12 11

The inOrder traversal is:
  2  4  5  6  7 11 12

The postOrder traversal is:
 2  5  4 11 12  7  6
```

665

Binary search tree with seven nodes



666

Review

- Four data structures
  - linked list
  - stacks
  - queues
  - trees
- Dynamic memory allocation
  - malloc()
  - free()

667

The End

- Thank you very much!

668