Chapter 1
Introduction to The C Language

Outline

1.1 Introduction

- We will learn the C programming language
  - Learn structured programming and proper programming techniques
- This course is appropriate for
  - Technically oriented people with little or no programming experience
  - Experienced programmers who want a deep and rigorous treatment of the language

1.6 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
+1400593419
+1200274027
```

  2. Assembly languages
     - English-like abbreviations representing elementary computer operations (translated via assemblers)
     - Example:

```
LOAD    BASEPAY
ADD     OVERPAY
STORE   GROSSPAY
```


1.6 Machine Languages, Assembly Languages, and High-level Languages (II)

- High-level languages
  - Similar to everyday English and use mathematical notations (translated via compilers)
  - Example:

```
grossPay = basePay + overTimePay
```

- **Interpreters**
- - directly execute program lines without compilation

1.7 History of C

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

1.8 The 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

1.9 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*

1.10 C and C++

- C++
  - Superset of C developed by Bjarne Stroustrup at Bell Labs
  - "Spruces up" C, and provides object-oriented capabilities
    - Objects - reuseable software components
  - Object-oriented design very powerful
    - 10 to 100 fold increase in productivity
  - Dominant language in industry and university
- Learning C++
  - Because C++ includes C, some feel it is best to master C, then learn C++
  - Chapter 15 of Deitel & Deitel begins an introduction to C++

1.12 Other High-level Languages

- A few other high-level languages have achieved broad acceptance

- FORTRAN
  - Scientific and engineering applications
- COBOL
  - Used to manipulate large amounts of data
- Pascal
  - Intended for academic use
- Java
  - Object-oriented; developed by Sun Microsystems
  - Good for creating web pages with dynamic and interactive content

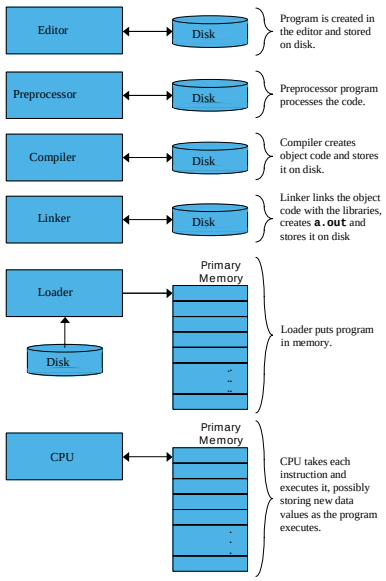
1.13 Structured Programming

- Structured programming
  - Disciplined approach to writing programs
  - Clear, easy to test and debug, and easy to modify
  - examples - Pascal and Ada

- Multitasking
  - Specifying that many activities run in parallel
  - Not supported directly by C and C++. Most operating systems now support multitasking.

1.14 Basics of a Typical C Program Development Environment

- Phases of C++ Programs:
  1. *Edit*
  2. *Preprocess*
  3. *Compile*
  4. *Link*
  5. *Load*
  6. *Execute*



1.18 General Notes About C

- Program clarity
  - Programs that are convoluted are difficult to read, understand, and modify
- C is a portable language
  - Programs can run on many different computers
  - However, portability is an elusive goal
- We will do a careful walkthrough of C
  - Some details and subtleties not covered
  - If you need additional technical details
    - Read the C standard document
    - Read other books and materials on C

Chapter 2
Introduction to C Programming

Outline

2.1 Introduction

- C programming language
  - Structured and disciplined approach to program design
- Structured programming
  - Introduced in chapters 3 and 4 of Deitel & Deitel
  - Used throughout our discussion on C programming

2.2 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  int main()
6  {
7     printf( "Welcome to C!\n" );
8
9     return 0;
10 }
```

```
Welcome to C!
```

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

2.2 A Simple C Program: Printing a Line of Text (II)

- **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 indicate a block
    - The bodies of all functions must be contained in braces

2.2 A Simple C Program: Printing a Line of Text (III)

- **`printf( "Welcome to C!\n" );`**
  - Instructs computer to perform an action
    - Specifically, prints 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

2.2 A Simple C Program: Printing a Line of Text (IV)

- **`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 misspelled, linker will spot error because it cannot find function in library

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

int main()
{
   int integer1, integer2, sum;        /* declaration */

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

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

Outline

1. Initialize variables
2. Input
2.1 Sum
3. Print

Program Output

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


2.3 Another Simple C Program: Adding Two Integers

- As before
  - Comments, **`#include <stdio.h>`** and **`main`**
- **`int integer1, integer2, sum;`**
  - Declaration of variables
    - Variables: locations in memory where a value can be stored
  - **`int`** means the variables can hold integers (**`-1`**, **`3`**, **`0`**, **`47`**)
  - **`integer1`**, **`integer2`**, **`sum`** - variable names (identifiers)
    - Identifiers: consist of letters, digits (cannot begin with a digit), and underscores, case sensitive
  - Declarations appear before executable statements
    - If not, syntax (compile) error

2.3 Another Simple C Program: Adding Two Integers (II)

- **`scanf( "%d", &integer1 );`**
  - Obtains value from user
    - **`scanf`** uses standard input (usually keyboard)
  - This **`scanf`** has two arguments
    - **`%d`** - indicates data should be a decimal integer
    - **`&integer1`** - location in memory to store variable
    - **`&`** is confusing in beginning - just remember to include it with the variable name in **`scanf`** statements
      - It will be discussed later

2.3 Another Simple C Program: Adding Two Integers (III)

- **`=`** (assignment operator)
  - Assigns value to a variable
  - Binary operator (has two operands)

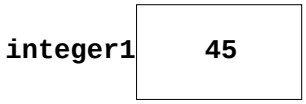
    `sum = variable1 + variable2;`

    `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 );
    ```


2.4 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



    change the)
- A visual representation

2.5 Arithmetic

- Arithmetic calculations are used in most programs
  - 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`**

2.5 Arithmetic (II)

- 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 mod s*	`r % s`

Operator(s)	Operation(s)	Order of evaluation (precedence)
`()`	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 re evaluated left to right.
`+` or `-`	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

2.6 Decision Making: Equality and Relational Operators

- **Executable statements**
  - Perform actions (calculations, input/output)
  - Perform decisions
    - May want to print "pass" or "fail" given the value of a test grade
- `if` control structure
  - Simple version in this section, more detail later
  - If a condition is true, body of **`if`** statement is 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

2.6 Decision Making: Equality and Relational Operators (II)

Standard algebraic equality operator or relational operator	C++ equality or relational operator	Example of C++ condition	Meaning of C++ condition
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**
Equality operators			
=	==	**x == y**	**x** is equal to **y**
≠	!=	**x != y**	**x** is not equal to **y**

```
1  /* Fig. 2.13: fig02_13.c
2     Using if statements, relational
3     operators, and equality operators */
4  #include <stdio.h>
5
6  int main()
7  {
8     int num1, num2;
9
10    printf( "Enter two integers, and I will tell you\n" );
11    printf( "the relationships they satisfy: " );
12    scanf( "%d%d", &num1, &num2  );   /* read two integers */
13
14    if ( num1 == num2 )
15       printf( "%d is equal to %d\n", num1, num2 );
16
17    if ( num1 != num2 )
18       printf( "%d is not equal to %d\n", num1, num2 );
19
20    if ( num1 < num2 )
21       printf( "%d is less than %d\n", num1, num2 );
22
23    if ( num1 > num2 )
24       printf( "%d is greater than %d\n", num1, num2 );
25
26    if ( num1 <= num2 )
27       printf( "%d is less than or equal to %d\n",
28               num1, num2 );
```

Outline

1. Declare variables

2. Input

2.1 `if` statements

3. Print

```
29
30    if ( num1 >= num2 )
31       printf( "%d is greater than or equal to %d\n",
32               num1, num2 );
33
34    return 0;   /* indicate program ended successfully */
35 }
```

Outline

3.1 Exit `main`

Program Output

```
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
```

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


2.6 Complete C 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

Chapter 3
Structured Program Development in C

Outline

3.1 Introduction

- Before writing a program:
  - Have a thorough understanding of problem
  - Carefully planned approach for solving it
- While writing a program:
  - Know what “building blocks” are available
  - Use good programming principles

3.2 Algorithms

- Computing problems
  - All can be solved by executing a series of actions in a specific order
- Algorithm: procedure in terms of
  - *Actions* to be executed
  - *Order* in which these actions are to be executed
- Program control
  - Specify order in which statements are to executed

3.3 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
    - Consists only of executable statements

3.4 Control Structures

- Sequential execution
  - 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`** led to many problems.
- Bohm and Jacopini
  - All programs written in terms of 3 control structures
    - Sequence structure: Built into C. Programs executed sequentially by default.
    - Selection structures: C has three types- **`if`**, **`if/else`**, and **`switch`** .
    - Repetition structures: Three types - **`while`**, **`do/while`**, and **`for`**.
    - No **goto** statements
    - These are C keywords

3.4 Control Structures (II)

- 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 beginning or end of a program, or a section of code (circles).
- Single-entry/single-exit control structures
  - *Control-structure stacking:* Connect exit point of one control structure to entry point of the next.
  - *Control structure nesting*: contain control structures can contain other control structures
  - Only 7 control structures in all; only 2 to combine them
  - Makes programs easy to build

3.5 The `if` Selection Structure

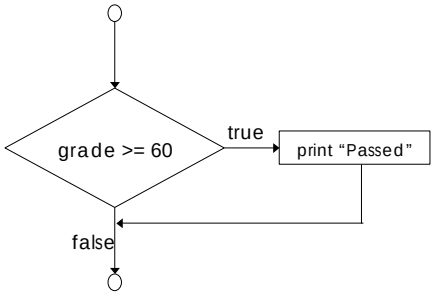
- Selection structure:
  - Used to choose among alternative courses of action
  - Pseudocode: *If student's grade is greater than or equal to 60*
    Print "Passed"
- If condition **`true`**
  - Print statement executed and program goes on to next statement.
  - If **`false`**, print statement is ignored and the program goes onto the next statement.
  - Indenting makes programs easier to read
    - C ignores whitespace characters.
- Pseudocode statement in C:

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

  - C code corresponds closely to the pseudocode

3.5 The `if` Selection Structure (II)

- 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
- **`if`** structure is a single-entry/single-exit structure.
- Still follows the general *action-decision* programming model



A decision can be made on any expression.

zero - **`false`**

nonzero - **`true`**

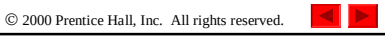
Example:

`3 - 4` is **`true`**

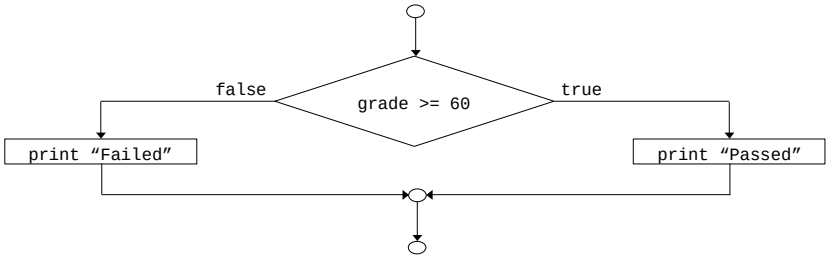
3.6 The `if/else` Selection Structure

- **`if`**
  - Only performs an action if the condition is **`true`**.
- **`if/else`**
  - A different action when condition is **`true`** than when condition is **`false`**
- Psuedocode: *If student's grade is greater than or equal to 60*
  Print "Passed"
  else
  Print "Failed"
  - Note spacing/indentation conventions
- C code:

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

3.6 The `if/else` Selection Structure (II)

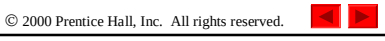


- Ternary conditional operator (**`?:`**)
  - Takes three arguments (condition, value if **`true`**, value if **`false`**)
  - Our pseudocode could be written:

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

OR

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

3.6 The `if/else` Selection Structure (III)

- Nested **`if/else`** structures
  - Test for multiple cases by placing **`if/else`** selection structures inside **`if/else`** selection structures

```
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"
```

  - Once condition is met, rest of statements skipped
  - Deep indentation usually not used in practice

3.6 The `if/else` Selection Structure (IV)

- 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,

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

    would be automatically executed

- Block: compound statements with declarations

3.6 The `if/else` Selection Structure (V)

- Syntax errors
  - Caught by compiler
- Logic errors:
  - Have their effect at execution time
  - Non-fatal: program runs, but has incorrect output
  - Fatal: program exits prematurely

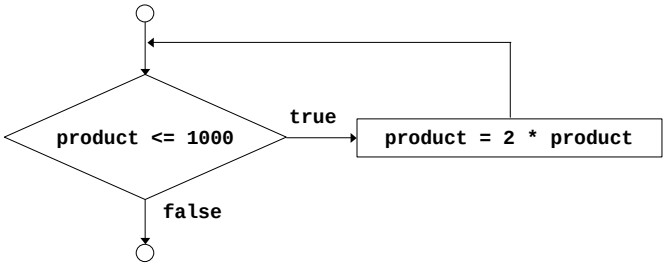
3.7 The `while` Repetition Structure

- 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`**

3.7 The `while` Repetition Structure (II)

- Example:

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



3.8 Formulating Algorithms (Counter-Controlled Repetition)

- 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

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

int main()
{
   int counter, grade, total, average;

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

   /* processing phase */
   while ( counter <= 10 ) {
      printf( "Enter grade: " );
      scanf( "%d", &grade );
      total = total + grade;
      counter = counter + 1;
   }

   /* termination phase */
   average = total / 10;
   printf( "Class average is %d\n", average );

   return 0;   /* indicate program ended successfully */
}
```

Outline

1. Initialize Variables
2. Execute Loop
3. Output results

Outline

Program Output

```
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
```

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement (Sentinel-Controlled Repetition)

- 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 when to end?
- Use sentinel value
  - Also called signal value, dummy value, or flag value
  - Indicates "end of data entry."
  - Loop ends when sentinel inputted
  - Sentinel value chosen so it cannot be confused with a regular input (such as **`-1`** in this case)
  - used for indefinite repetition

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement (Sentinel-Controlled Repetition) (II)

- 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
  - This helps the breakup of programs for top-down refinement

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement (III)

- 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)*
- 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"*

```
1  /* Fig. 3.8: fig03_08.c
2     Class average program with
3     sentinel-controlled repetition */
4  #include <stdio.h>
5
6  int main()
7  {
8     float average;              /* new data type */
9     int counter, grade, total;
10
11    /* initialization phase */
12    total = 0;
13    counter = 0;
14
15    /* processing phase */
16    printf( "Enter grade, -1 to end: " );
17    scanf( "%d", &grade );
18
19    while ( grade != -1 ) {
20       total = total + grade;
21       counter = counter + 1;
22       printf( "Enter grade, -1 to end: " );
23       scanf( "%d", &grade );
24    }
```

Outline

1. Initialize Variables

2. Get user input

2.1 Perform Loop

```
25
26    /* termination phase */
27    if ( counter != 0 ) {
28       average = ( float ) total / counter;
29       printf( "Class average is %.2f", average );
30    }
31    else
32       printf( "No grades were entered\n" );
33
34    return 0;   /* indicate program ended successfully */
35 }
```

Outline

3. Calculate Average

3.1 Print Results

Program Output

```
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
```


3.10 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

3.10 Nested control structures (II)

- 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

3.10 Nested control structures (III)

- 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"

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

int main()
{
   /* initializing variables in declarations */
   int passes = 0, failures = 0, student = 1, result;

   /* process 10 students; counter-controlled loop */
   while ( student <= 10 ) {
      printf( "Enter result ( 1=pass,2=fail ): " );
      scanf( "%d", &result );

      if ( result == 1 )        /* if/else nested in while */
         passes = passes + 1;
      else
         failures = failures + 1;

      student = student + 1;
   }

   printf( "Passed %d\n", passes );
   printf( "Failed %d\n", failures );

   if ( passes > 8 )
      printf( "Raise tuition\n" );

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

Outline

1. Initialize variables

2. Input data and count passes/failures

3. Print results

```
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
```

Outline

Program Output

3.11 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)
```


3.12 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, then the expression it is in is evaluated
- Postincrement
  - Operator is used after the variable (`c++` or `c--`)
  - Expression executes, then the variable is changed
- If `c = 5`, then
  `printf( "%d", ++c);`
    - Prints `6`
  `printf( "%d", c++);`
    - Prints `5`
  - In either case, `c` now has the value of `6`

3.12 Increment and Decrement Operators (II)

- When variable is not inside an expression
  - Preincrementing and postincrementing have the same effect.

```
++c;
cout << c;
```

and

```
c++;
cout << c;
```

have the same effect.

Chapter 4 - Program Control

Outline

4.1 Introduction

- Should feel comfortable writing simple C programs
- Coverage
  - Repetition, in greater detail
  - Logical operators for combining conditions
  - Principles of structured programming

4.2 The Essentials of Repetition

- Loop
  - Group of instructions computer executes repeatedly while some condition remains true
- Counter-controlled repetition
  - Definite repetition - know how many times loop will execute
  - Control variable used to count repetitions
- Sentinel-controlled repetition
  - Indefinite repetition
  - Used when number of repetitions not known
  - Sentinel value usually indicates "end of data"

4.3 Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires
  - *name* of a control variable (or loop counter).
  - *initial value* of the control variable.
  - condition that tests for the *final value* of the control variable (i.e., whether looping should continue).
  - *increment* (or *decrement*) by which the control variable is modified each time through the loop.

4.3 Essentials of Counter-Controlled Repetition (II)

- Example:

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

  - **`int counter = 1;`** names **`counter`**, declares it to be an integer, reserves space for it in memory, and sets it to an initial value of **`1`**

4.4 The `for` Repetition Structure

- Format when using **`for`** loops

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

  – good for counter-controlled repetition

Example:

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

No semicolon after last expression

– Prints the integers from one to ten.

4.4 The `for` Repetition Structure (II)

- **`For`** loops can usually be rewritten as **`while`** loops:

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

- Initialization and increment
  – Can be comma-separated lists

```
for (int i = 0, j = 0;  j + i <= 10; j++, i++)
  printf( "%d\n", j + i );
```


4.5 The For Structure: Notes and Observations

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

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

    is equivalent to

```
for ( j = 2; j <= 80; j += 5 )
```

- "Increment" may be negative (decrement)
- If loop continuation condition initially **`false`**
  - Body of **`for`** structure not performed
  - Control proceeds with statement after **`for`** structure

4.5 The For Structure: Notes and Observations (II)

- Control variable
  - Often printed or used inside **`for`** body, but not necessary
- **`for`** flowcharted like **`while`**
- **`often used for infinite loops`**

```
1  /* Fig. 4.5: fig04_05.c
2     Summation with for */
3  #include <stdio.h>
4
5  int main()
6  {
7     int sum = 0, number;
8
9     for ( number = 2; number <= 100; number += 2 )
10       sum += number;
11
12    printf( "Sum is %d\n", sum );
13
14    return 0;
15 }
```

```
Sum is 2550
```

Outline

4.6 Examples Using the `for` Structure

Program to sum the even numbers from 2 to 100

Program Output

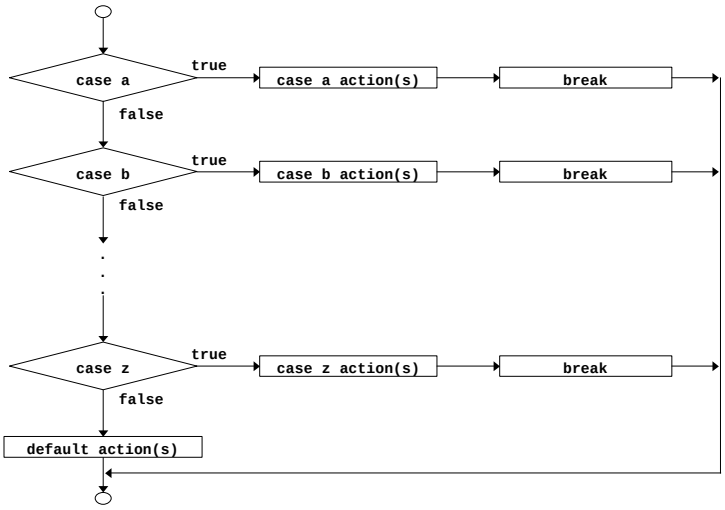
4.7 The `switch` Multiple-Selection Structure

- **`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;`** causes exit from structure

4.7 The `switch` Multiple-Selection Structure (II)




```
1  /* Fig. 4.7: fig04_07.c
2     Counting letter grades */
3  #include <stdio.h>
4
5  int main()
6  {
7     int grade;
8     int aCount = 0, bCount = 0, cCount = 0,
9         dCount = 0, fCount = 0;
10
11    printf(  "Enter the letter grades.\n"  );
12    printf(  "Enter the EOF character to end input.\n"  );
13
14    while ( ( grade = getchar() ) != EOF ) {
15
16       switch ( grade ) {    /* switch nested in while */
17
18          case 'A': case 'a':  /* grade was uppercase A */
19             ++aCount;         /* or lowercase a */
20             break;
21
22          case 'B': case 'b':  /* grade was uppercase B */
23             ++bCount;         /* or lowercase b */
24             break;
25
26          case 'C': case 'c':  /* grade was uppercase C */
27             ++cCount;         /* or lowercase c */
28             break;
29
30          case 'D': case 'd':  /* grade was uppercase D */
31             ++dCount;         /* or lowercase d */
32             break;
```

Outline

1. Initialize variables

2. Input data

2.1 Use switch loop to update count

```
33
34          case 'F': case 'f':  /* grade was uppercase F */
35             ++fCount;         /* or lowercase f */
36             break;
37
38          case '\n': case' ':  /* ignore these in input */
39             break;
40
41          default:        /* catch all other characters */
42             printf( "Incorrect letter grade entered." );
43             printf( " Enter a new grade.\n" );
44             break;
45       }
46    }
47
48    printf( "\nTotals for each letter grade are:\n" );
49    printf( "A: %d\n", aCount );
50    printf( "B: %d\n", bCount );
51    printf( "C: %d\n", cCount );
52    printf( "D: %d\n", dCount );
53    printf( "F: %d\n", fCount );
54
55    return 0;
56 }
```

Outline

2.1 Use switch loop to update count

3. Print results

```
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

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Outline

Program Output

4.8 The do/while Repetition Structure

- The **do/while** repetition structure
  - 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* **);**
  - Good practice to put brackets in, even if not required

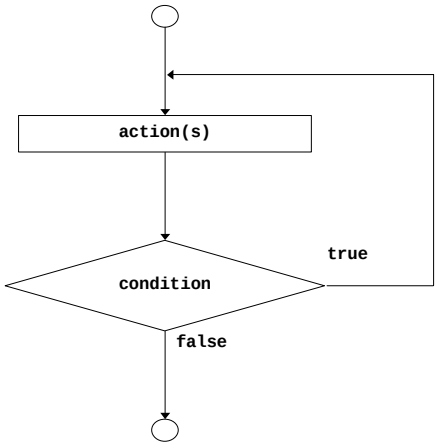
4.8 The do/while Repetition Structure (II)

- Example (letting **counter = 1**)

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

Prints the integers from 1 to 10

4.8 The do/while Repetition Structure (III)




```
1  /* Fig. 4.9: fig04_09.c
2     Using the do/while repetition structure */
3  #include <stdio.h>
4
5  int main()
6  {
7     int counter = 1;
8
9     do {
10       printf( "%d  ", counter );
11    } while ( ++counter <= 10 );
12
13    return 0;
14 }
```

Outline

1. Initialize variable

2. Loop

3. Print

Program Output

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

4.9 The `break` and `continue` Statements

- **`break`**
  - Causes immediate exit from a **`while`**, **`for`**, **`do/while`** or **`switch`** structure
  - 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`** structure

4.9 The `break` and `continue` Statements (II)

- **`continue`**
  - Skips the remaining statements in the body of a **`while`**, **`for`** or **`do/while`** structure
    - 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`** structure
    - Increment expression is executed, then the loop-continuation test is evaluated

```
1  /* Fig. 4.12: fig04_12.c
2     Using the continue statement in a for structure */
3  #include <stdio.h>
4
5  int main()
6  {
7     int x;
8
9     for ( x = 1; x <= 10; x++ ) {
10
11       if ( x == 5 )
12          continue;  /* skip remaining code in loop only
13                        if x == 5 */
14
15       printf( "%d ", x );
16    }
17
18    printf( "\nUsed continue to skip printing the value 5\n" );
19    return 0;
20 }
```

Outline

1. Initialize variable

2. Loop

3. Print

Program Output

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


4.10 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

4.11 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:

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

Checks paycode, if it is 4 then a bonus is awarded

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

- 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

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

- 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;`
  - lvalues can be used as rvalues, but not vice versa
    - `y = x;`

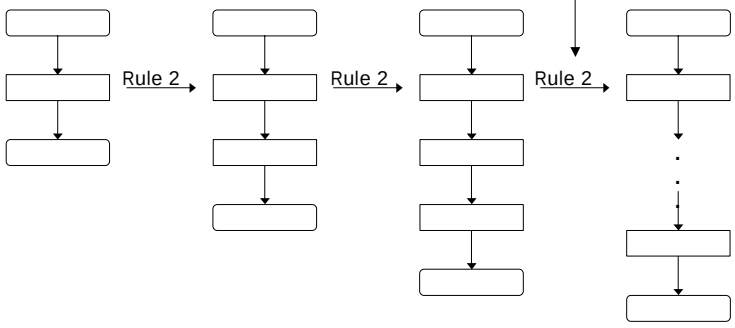
4.12 Structured-Programming Summary

- 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) Any rectangle (action) can be replaced by two rectangles (actions) in sequence.
    3) 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.

4.12 Structured-Programming Summary (II)

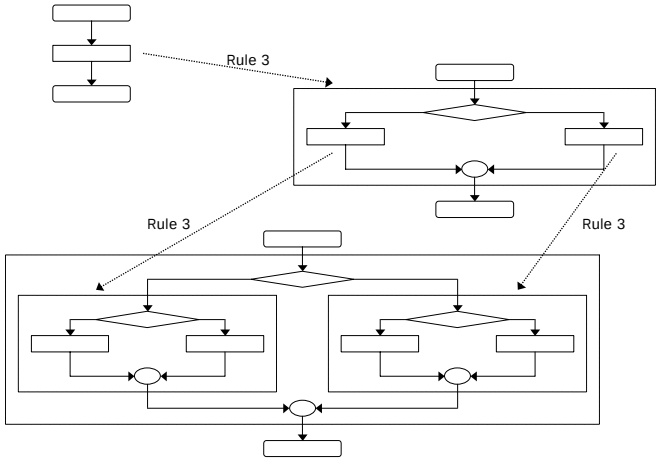
Rule 1 - Begin with the simplest flowchart

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



4.12 Structured-Programming Summary (III)

Rule 3 - Replace any rectangle with a control structure



4.12 Structured-Programming Summary (IV)

- All programs can be broken down into 3 parts

  Sequence - trivial

  Selection - **`if`**, **`if/else`**, or **`switch`**

  Repetition - **`while`**, **`do/while`**, or **`for`**

  – Any selection can be rewritten as an **`if`** statement, and any repetition can be rewritten as a **`while`** statement

- Programs are reduced to
  – Sequence
  – **`if`** structure (selection)
  – **`while`** structure (repetition)
    - The control structures can only be combined in two ways- nesting (rule 3) and stacking (rule 2)
  – This promotes simplicity

Chapter 5 - Functions

Outline

5.1 Introduction

- Divide and conquer
  - Construct a program from smaller pieces or components
  - Each piece more manageable than the original program

5.2 Program Modules in C

- Functions
  - Modules in C
  - Programs written by combining user-defined functions with library functions
    - C standard library has a wide variety of functions
    - Makes programmer's job easier - avoid reinventing the wheel
- Function calls
  - Invoking functions
    - Provide function name and arguments (data)
    - Function performs operations or manipulations
    - Function returns results
  - Boss asks worker to complete task
    - Worker gets information, does task, returns result
    - Information hiding: boss does not know details

5.3 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

5.4 Functions

- Functions
  - **Modularize a program**
  - Variables declared inside functions are local variables
    - Known only in function defined
  - **Parameters**
    - Communicate information between functions
    - Local variables
- **Benefits**
  - **Divide and conquer**
    - Manageable program development
  - **Software reusability**
    - Use existing functions as building blocks for new programs
    - Abstraction - hide internal details (library functions)
  - **Avoids code repetition**

5.5 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`** - function returns nothing
– Parameter-list: comma separated list, declares parameters (default **`int`**)

5.5 Function Definitions (II)

- Function definition format (continued)

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

– **Declarations and statements**: function body (block)
  • Variables can be declared inside blocks (can be nested)
  • Function can not be defined inside another function
– **Returning control**
  • If nothing returned
    – **`return;`**
    – or, until reaches right brace
  • If something returned
    – **`return`** *expression* **`;`**

```c
/* Fig. 5.4: fig05_04.c
   Finding the maximum of three integers */
#include <stdio.h>

int maximum( int, int, int );   /* function prototype */

int main()
{
   int a, b, c;

   printf( "Enter three integers: " );
   scanf( "%d%d%d", &a, &b, &c );
   printf( "Maximum is: %d\n", maximum( a, b, c ) );

   return 0;
}

/* Function maximum definition */
int maximum( int x, int y, int z )
{
   int max = x;

   if ( y > max )
      max = y;

   if ( z > max )
      max = z;

   return max;
}
```

```
Enter three integers: 22 85 17
Maximum is: 85
```


Outline

1. Function prototype (3 parameters)
2. Input values

2.1 Call function

- Function definition

5.6 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

    ```
    int maximum( int, int, int );
    ```

    - Takes in 3 **`int`**s
    - Returns an **`int`**
- Promotion rules and conversions
  - Converting to lower types can lead to errors

5.7 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

5.8 Calling Functions: Call by Value and Call by Reference

- Used when invoking functions
- 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

5.9 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 every time program is executed
- 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) /* no. b/w 1 and  6*/
    ```


5.9 Random Number Generation (II)

- **srand** function
  - **<stdlib.h>**
  - Takes an integer seed - jumps to location in "random" sequence

    ```
    srand( seed );
    ```

  - **srand(time(NULL)); /* load <time.h> */**
    - **time(NULL)-** time 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>

int main()
{
   int i;
   unsigned seed;

   printf( "Enter seed: " );
   scanf( "%u", &seed );
   srand( seed );

   for ( i = 1; i <= 10; i++ ) {
      printf( "%10d", 1 + ( rand() % 6 ) );

      if ( i % 5 == 0 )
         printf( "\n" );
   }

   return 0;
}
```

Outline

1. Initialize seed

2. Input value for seed

2.1 Use `srand` to change random sequence

2.2 Define Loop

3. Generate and output random numbers

```
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
```

Outline

Program Output

5.10 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
  - Player loses if he rolls 7 before rolling his point

```
1  /* Fig. 5.10: fig05_10.c
2     Craps */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  int rollDice( void );
8
9  int main()
10 {
11    int gameStatus, sum, myPoint;
12
13    srand( time( NULL ) );
14    sum = rollDice();           /* first roll of the dice */
15
16    switch ( sum ) {
17       case 7: case 11:         /* win on first roll */
18          gameStatus = 1;
19          break;
20       case 2: case 3: case 12:  /* lose on first roll */
21          gameStatus = 2;
22          break;
23       default:                 /* remember point */
24          gameStatus = 0;
25          myPoint = sum;
26          printf( "Point is %d\n", myPoint );
27          break;
28    }
29
30    while ( gameStatus == 0 ) {    /* keep rolling */
31       sum = rollDice();
32
```

Outline

1. `rollDice` prototype

1.1 Initialize variables

1.2 Seed `srand`

2. Define `switch` statement for win/loss/continue

2.1 Loop

```
33       if ( sum == myPoint )       /* win by making point */
34          gameStatus = 1;
35       else
36          if ( sum == 7 )          /* lose by rolling 7 */
37             gameStatus = 2;
38    }
39
40    if ( gameStatus == 1 )
41       printf( "Player wins\n" );
42    else
43       printf( "Player loses\n" );
44
45    return 0;
46 }
47
48 int rollDice( void )
49 {
50    int die1, die2, workSum;
51
52    die1 = 1 + ( rand() % 6 );
53    die2 = 1 + ( rand() % 6 );
54    workSum = die1 + die2;
55    printf( "Player rolled %d + %d = %d\n", die1, die2, workSum );
56    return workSum;
57 }
```

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

Outline

2.2 Print win/loss

```
Player rolled 6 + 6 = 12
Player loses
```

Outline

Program Output

```
Player rolled 4 + 6 = 10
Point is 10
Player rolled 2 + 4 = 6
Player rolled 6 + 5 = 11
Player rolled 3 + 3 = 6
Player rolled 6 + 4 = 10
Player wins
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 1 + 4 = 5
Player rolled 5 + 4 = 9
Player rolled 4 + 6 = 10
Player rolled 6 + 3 = 9
Player rolled 1 + 2 = 3
Player rolled 5 + 2 = 7
Player loses
```

5.11 Storage Classes

- Storage class specifiers
  - Storage duration - how long an object exists in memory
  - Scope - where object can be referenced in program
  - Linkage - what files 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;`**

5.11 Storage Classes (II)

- 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

5.12 Scope Rules

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

5.12 Scope Rules (II)

- Block scope
  - Identifier declared inside a block
    - Block scope begins at declaration, ends at right brace
  - Variables, function parameters (local variables of function)
  - Outer blocks "hidden" from inner blocks if same variable name
- Function prototype scope
  - Identifiers in parameter list
  - Names in function prototype optional, and can be used anywhere

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

void a( void );   /* function prototype */
void b( void );   /* function prototype */
void c( void );   /* function prototype */

int x = 1;        /* global variable */

int main()
{
   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;

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

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

   a();         /* a has automatic local x */
   b();         /* b has static local x */
   c();         /* c uses global x */
   a();         /* a reinitializes automatic local x */
   b();         /* static local x retains its previous value */
   c();         /* global x also retains its value */
```

Outline

1. Function prototypes

1.1 Initialize global variable

1.2 Initialize local variable

1.3 Initialize local variable in block

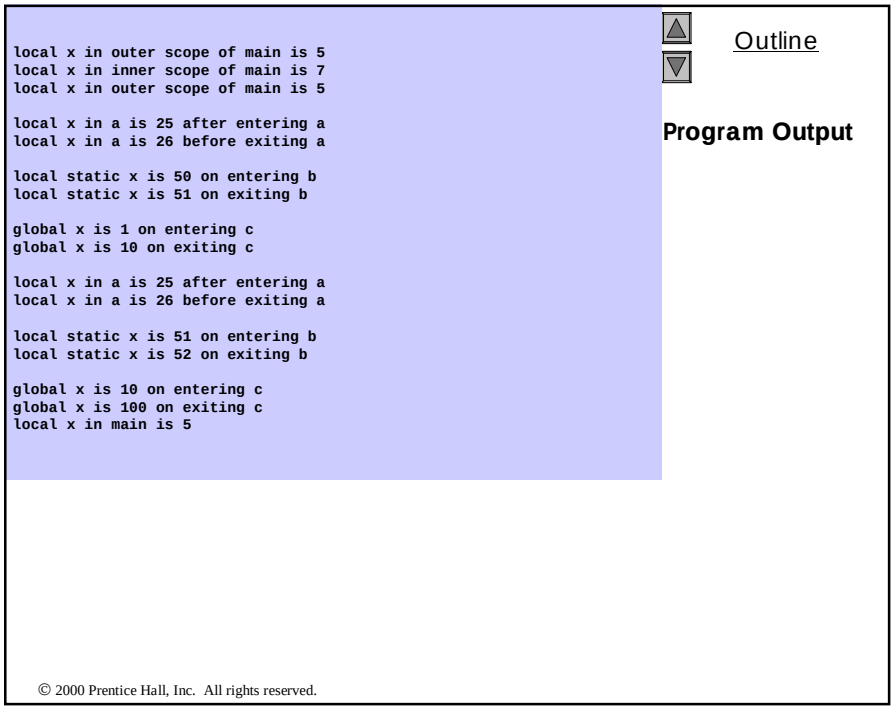
2. Call functions

3. Output results

```c

   printf( "local x in main is %d\n", x );
   return 0;
}

void a( void )
{
   int x = 25;  /* initialized each time a is called */

   printf( "\nlocal x in a is %d after entering a\n", x );
   ++x;
   printf( "local x in a is %d before exiting a\n", x );
}

void b( void )
{
    static int x = 50;  /* static initialization only */
                        /* first time b is called */
    printf( "\nlocal static x is %d on entering b\n", x );
    ++x;
    printf( "local static x is %d on exiting b\n", x );
}

void c( void )
{
   printf( "\nglobal x is %d on entering c\n", x );
   x *= 10;
   printf( "global x is %d on exiting c\n", x );
}
```

Outline

3.1 Function definitions



5.13 Recursion

- Recursive functions
  - Function that calls itself
  - Can only solve a base case
  - Divides up problem into
    - What it can do
    - What it cannot do - resembles original problem
      - Launches a new copy of itself (recursion step)
- Eventually base case gets solved
  - Gets plugged in, works its way up and solves whole problem

5.13 Recursion (II)

- Example: factorial:

  `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;`

5.14 Example Using Recursion: The Fibonacci Series

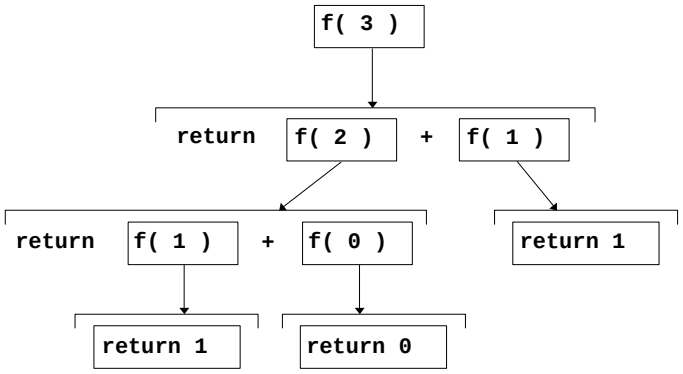
- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...

  – Each number sum of the previous two

  `fib(n) = fib(n-1) + fib(n-2)` - recursive formula

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


5.14 Example Using Recursion: The Fibonacci Series (II)




```
1  /* Fig. 5.15: fig05_15.c
2     Recursive fibonacci function */
3  #include <stdio.h>
4
5  long fibonacci( long );
6
7  int main()
8  {
9     long result, number;
10
11    printf( "Enter an integer: " );
12    scanf( "%ld", &number );
13    result = fibonacci( number );
14    printf( "Fibonacci( %ld ) = %ld\n", number, result );
15    return 0;
16 }
17
18 /* Recursive definition of function fibonacci */
19 long fibonacci( long n )
20 {
21    if ( n == 0 || n == 1 )
22       return n;
23    else
24       return fibonacci( n - 1 ) + fibonacci( n - 2 );
25 }
```

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

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

Outline

1. Function prototype

1.1 Initialize variables

2. Input an integer

2.1 Call function `fibonacci`

2.2 Output results.

3. Define `fibonacci` recursively

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

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

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
```

Outline

Program Output

5.15 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)

Chapter 6 - Arrays

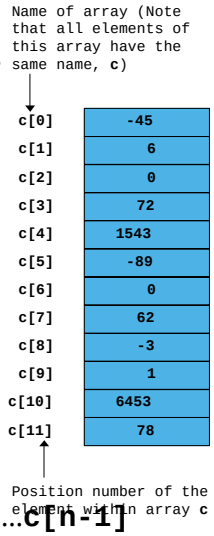
Outline

6.1 Introduction

- *Arrays*
  - Structures of related data items
  - Static entity - same size throughout program
  - Dynamic data structures discussed in Chapter 12

6.2 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]`**



6.2 Arrays (II)

- Array elements are like normal variables

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

  - Perform operations in subscript.

If **`x = 3`**,

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

6.3 Declaring Arrays

- When declaring arrays, specify
  - Name
  - Type of array
  - Number of elements

```
arrayType arrayName[
 numberOfElements ];
int c[ 10 ];
float myArray[ 3284 ];
```

- Declaring multiple arrays of same type
  - Format similar to regular variables

```
int b[ 100 ], x[ 27 ];
```


6.4 Examples Using Arrays

- Initializers

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

  - If not enough initializers, rightmost elements become **0**
  - If too many, syntax error

```
int n[5] = {0}
```

    - All elements **0**
  - 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

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

int main()
{
   int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
   int i, j;

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

   for ( i = 0; i <= SIZE - 1; i++ ) {
      printf( "%7d%13d        ", i, n[ i ]) ;

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

      printf( "\n" );
   }

   return 0;
}
```

Outline

1. Initialize array

2. Loop

3. Print


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

Outline

Program Output

6.4 Examples Using Arrays (II)

- Character arrays
  - String **"hello"** 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

```
char string1[] = { 'f', 'i', 'r', 's',
  't', '\0' };
```


6.4 Examples Using Arrays (III)

- Character arrays (continued)
  - 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

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

int main()
{
   char string1[ 20 ], string2[] = "string literal";
   int i;

   printf(" Enter a string: ");
   scanf( "%s", string1 );
   printf( "string1 is: %s\nstring2: is %s\n"
           "string1 with spaces between characters is:\n",
           string1, string2 );

   for ( i = 0; string1[ i ] != '\0'; i++ )
      printf( "%c ", string1[ i ] );

   printf( "\n" );
   return 0;
}
```

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

Outline

1. Initialize strings
2. Print strings
   2.1 Define loop
   2.2 Print characters individually
   2.3 Input string
3. Print string

6.5 Passing Arrays to Functions

- Passing arrays
  - Specify array name without 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

6.5 Passing Arrays to Functions (II)

- Function prototype

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

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

```
1  /* Fig. 6.13: fig06_13.c
2     Passing arrays and individual array elements to functions */
3  #include <stdio.h>
4  #define SIZE 5
5
6  void modifyArray( int [], int );  /* appears strange */
7  void modifyElement( int );
8
9  int main()
10 {
11    int a[ SIZE ] = { 0, 1, 2, 3, 4 }, i;
12
13    printf( "Effects of passing entire array call "
14            "by reference:\n\nThe values of the "
15            "original array are:\n" );
16
17    for ( i = 0; i <= SIZE - 1; i++ )
18       printf( "%3d", a[ i ] );
19
20    printf( "\n" );
21    modifyArray( a, SIZE );  /* passed call by reference */
22    printf( "The values of the modified array are:\n" );
23
24    for ( i = 0; i <= SIZE - 1; i++ )
25       printf( "%3d", a[ i ] );
26
27    printf( "\n\n\nEffects of passing array element call "
28            "by value:\n\nThe value of a[3] is %d\n", a[ 3 ] );
29    modifyElement( a[ 3 ] );
30    printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );
31    return 0;
32 }
```

Entire arrays passed call-by-reference, and can be modified

Array elements passed call-by-value, and cannot be modified

Outline

1. Function definitions

2. Pass array to a function

2.1 Pass array

o a

3. Print

```
33
34 void modifyArray( int b[], int size )
35 {
36    int j;
37
38    for ( j = 0; j <= size - 1; j++ )
39       b[ j ] *= 2;
40 }
41
42 void modifyElement( int e )
43 {
44    printf( "Value in modifyElement is %d\n", e *= 2 );
45 }
```

```
Effects of passing entire array call 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 call by value:


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

Outline

3.1 Function definitions

Program Output

6.6 Sorting Arrays

- Sorting data
  - Important computing application
  - Virtually every organization must sort some data
    - Massive amounts must be sorted
- 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

6.7 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

```
1  /* Fig. 6.16: fig06_16.c
2     This program introduces the topic of survey data analysis.
3     It computes the mean, median, and  mode of the data */
4  #include <stdio.h>
5  #define SIZE 99
6
7  void mean( const int [] );
8  void median( int [] );
9  void mode( int [], const int [] ) ;
10 void bubbleSort( int [] );
11 void printArray( const int [] );
12
13 int main()
14 {
15    int frequency[ 10 ] = { 0 };
16    int response[ SIZE ] =
17       { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
18         7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
19         6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
20         7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
21         6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
22         7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
23         5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
24         7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
25         7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
26         4, 5, 6, 1, 6, 5, 7, 8, 7 };
27
28    mean( response );
29    median( response );
30    mode( frequency, response );
31    return 0;
32 }
```

Outline

1. Function prototypes

1.1 Initialize array

2. Call functions `mean`, `median`, and `mode`

```
33
34 void mean( const int answer[] )
35 {
36    int j, total = 0;
37
38    printf( "%s\n%s\n%s\n", "********", "  Mean", "********" );
39
40    for ( j = 0; j <= SIZE - 1; j++ )
41       total += answer[  j  ];
42
43    printf( "The mean is the average value of the data\n"
44            "items. The mean is equal to the total of\n"
45            "all the data items divided by the number\n"
46            "of data items ( %d ). The mean value for\n"
47            "this run is: %d / %d = %.4f\n\n",
48            SIZE, total, SIZE, (  double  ) total / SIZE );
49 }
50
51 void median( int answer[] )
52 {
53    printf( "\n%s\n%s\n%s\n%s",
54            "********", " Median", "********",
55            "The unsorted array of responses is" );
56
57    printArray( answer );
58    bubbleSort( answer );
59    printf( "\n\nThe sorted array is" );
60    printArray( answer );
61    printf( "\n\nThe median is element %d of\n"
62            "the sorted %d element array.\n"
63            "For this run the median is %d\n\n",
64            SIZE / 2, SIZE, answer[ SIZE / 2 ] );
```

Outline

3. Define function `mean`

3.1 Define function `median`

3.1.1 Sort Array

3.1.2 Print middle element

```
65 }
66
67 void mode( int freq[], const int answer[] )
68 {
69    int rating, j, h, largest = 0, modeValue = 0;
70
71    printf( "\n%s\n%s\n%s\n",
72            "********", "  Mode", "********" );
73
74    for ( rating = 1; rating <= 9; rating++ )
75       freq[ rating ] = 0;
76
77    for ( j = 0; j <= SIZE - 1; j++ )
78       ++freq[ answer[ j ] ];
79
80    printf( "%s%11s%19s\n\n%54s\n%54s\n\n",
81            "Response", "Frequency", "Histogram",
82            "1    1    2    2", "5    0    5    0    5" );
83
84    for ( rating = 1; rating <= 9; rating++ ) {
85       printf( "%8d%11d          ", rating, freq[ rating ] );
86
87       if ( freq[ rating ] > largest ) {
88          largest = freq[ rating ];
89          modeValue = rating;
90       }
91
92       for ( h = 1; h <= freq[ rating ]; h++ )
93          printf( "*" );
94
```

Notice how the subscript in **`frequency[]`** is the value of an element in **`response[]`** (**`answer[]`**)

Print stars depending on value of **`frequency[]`**

Outline

3.2 Define function `mode`

3.2.1 Increase `frequency[]` depending on `response[]`

```
95       printf( "\n" );
96    }
97
98    printf( "The mode is the most frequent value.\n"
99            "For this run the mode is %d which occurred"
100           " %d times.\n", modeValue, largest );
101}
102
103void bubbleSort( int a[] )
104{
105   int pass, j, hold;
106
107   for ( pass = 1; pass <= SIZE - 1; pass++ )
108
109      for ( j = 0; j <= SIZE - 2; j++ )
110
111         if ( a[ j ] > a[ j + 1 ] ) {
112            hold = a[ j ];
113            a[ j ] = a[ j + 1 ];
114            a[ j + 1 ] = hold;
115         }
116}
117
118void printArray( const int a[] )
119{
120   int j;
121
122   for ( j = 0; j <= SIZE - 1; j++ ) {
123
124      if ( j % 20 == 0 )
125         printf( "\n" );
```

Bubble sort: if elements are out of order, swap them.

Outline

3.3 Define `bubbleSort`

3.3 Define `printArray`

```
126
127      printf( "%2d", a[ j ] );
128   }
129}
```

Outline

Program Output

```
********
 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
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

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.
```

Outline

Program Output

6.8 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

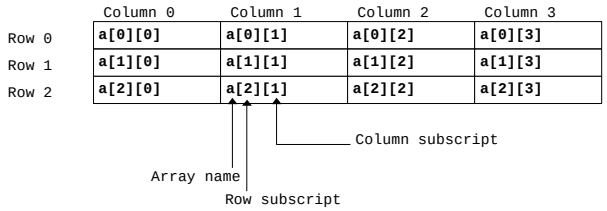
6.8 Searching Arrays: Linear Search and Binary Search (II)

- 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

      $2^5 > 30$

6.9 Multiple-Subscripted Arrays

- Multiple subscripted arrays
  - Tables with rows and columns (m by n array)
  - Like matrices: specify row, then column



6.9 Multiple-Subscripted Arrays (II)

- Initialization

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

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 } };
```

1	0
3	4

- Referencing elements
  - Specify row, then column

```
printf( "%d", b[ 0 ][ 1 ] );
```


```
1  /* Fig. 6.22: fig06_22.c
2     Double-subscripted array example */
3  #include <stdio.h>
4  #define STUDENTS 3
5  #define EXAMS 4
6
7  int minimum( const int [][ EXAMS ], int, int );
8  int maximum( const int [][ EXAMS ], int, int );
9  double average( const int [], int );
10 void printArray( const int [][ EXAMS ], int, int );
11
12 int main()
13 {
14    int student;
15    const int studentGrades[ STUDENTS ][ EXAMS ] =
16       { { 77, 68, 86, 73 },
17         { 96, 87, 89, 78 },
18         { 70, 90, 86, 81 } };
19
20    printf( "The array is:\n" );
21    printArray( studentGrades, STUDENTS, EXAMS );
22    printf( "\n\nLowest grade: %d\nHighest grade: %d\n",
23            minimum( studentGrades, STUDENTS, EXAMS ),
24            maximum( studentGrades, STUDENTS, EXAMS ) );
25
26    for ( student = 0; student <= STUDENTS - 1; student++ )
27       printf( "The average grade for student %d is %.2f\n",
28               student,
29               average( studentGrades[ student ], EXAMS ) );
30
31    return 0;
32 }
```

Each row is a particular student, each column is the grades on the exam.

Outline

1. Initialize variables

1.1 Define ... to take ...ipted arrays

1.2 Initialize `studentgrades[ ][]`

2. Call functions `minimum`, `maximum`, and `average`

```
33
34 /* Find the minimum grade */
35 int minimum( const int grades[][ EXAMS ],
36              int pupils, int tests )
37 {
38    int i, j, lowGrade = 100;
39
40    for ( i = 0; i <= pupils - 1; i++ )
41       for ( j = 0; j <= tests - 1; j++ )
42          if ( grades[ i ][ j ] < lowGrade )
43             lowGrade = grades[ i ][ j ];
44
45    return lowGrade;
46 }
47
48 /* Find the maximum grade */
49 int maximum( const int grades[][ EXAMS ],
50              int pupils, int tests )
51 {
52    int i, j, highGrade = 0;
53
54    for ( i = 0; i <= pupils - 1; i++ )
55       for ( j = 0; j <= tests - 1; j++ )
56          if ( grades[ i ][ j ] > highGrade )
57             highGrade = grades[ i ][ j ];
58
59    return highGrade;
60 }
61
62 /* Determine the average grade for a particular exam */
63 double average( const int setOfGrades[], int tests )
64 {
```

Outline

3. Define functions

```
65    int i, total = 0;
66
67    for ( i = 0; i <= tests - 1; i++ )
68       total += setOfGrades[ i ];
69
70    return ( double ) total / tests;
71 }
72
73 /* Print the array */
74 void printArray( const int grades[][ EXAMS ],
75                  int pupils, int tests )
76 {
77    int i, j;
78
79    printf( "                [0]  [1]  [2]  [3]" );
80
81    for ( i = 0; i <= pupils - 1; i++ ) {
82       printf( "\nstudentGrades[%d] ", i );
83
84       for ( j = 0; j <= tests - 1; j++ )
85          printf( "%-5d", grades[ i ][ j ] );
86    }
87 }
```

Outline

3. Define functions

Outline

Program Output

```
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
```